NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A VLSI DESIGN OF A
RADIX-4 FLOATING POINT
FFT BUTTERFLY

by

Michael Lee Zimmer

December, 1991

Thesis Advisor: Herschel H. Loomis, Jr.

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) EC	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
A VLSI Design of a Radix-4 Floating Point FFT Butterfly

12. PERSONAL AUTHOR(S) ZIMMER, Michael L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	December 1991	126

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Digital Arithmetic; FFT Butterfly Design; Cyclic Spectrum Analysis; Genesil Silicon Compiler

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

Cyclic Spectrum Analysis is used to exploit the cyclostationary properties of signals and systems. Implementing such a system will require high speed arithmetic processing. Investigations into high speed arithmetic and FFT design are conducted. Integrated circuits of a 45 MHz floating point multiplier, adder, and rate-1/4 radix-4 FFT butterfly implemented with a 20-bit word size, are presented using the Genesil Silicon Compiler.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
XX UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL LOOMIS, Herschel H., Jr.	22b. TELEPHONE (*Include Area code*) (408) 646-3124	22c. OFFICE SYMBOL EC/Lm

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

A VLSI DESIGN OF A RADIX-4 FLOATING POINT FFT BUTTERFLY

by

Michael Lee Zimmer
Lieutenant, United States Navy
B.S., Kearney State College, 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1991

ABSTRACT

Cyclic Spectrum Analysis is used to exploit the cyclostationary properties of signals and systems. Implementing such a system will require high speed arithmetic processing. Investigations into high speed arithmetic and FFT design are conducted. Integrated circuits of a 45 MHz floating point multiplier, adder, and rate-1/4 radix-4 FFT butterfly, implemented with a 20-bit word size, are presented using the Genesil Silicon Compiler.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. Cyclic Spectrum Analysis

Cyclic Spectrum Analysis is used to investigate cyclostationary properties of signals and systems. This technique generalizes conventional spectral analysis to include periodic time variant signals and systems. Cyclic spectrum analysis is well suited for signal detection, modulation recognition, signal parameter estimation and the design of communications systems. Applications to spaceborne systems are possible if the integrated circuit (IC) is radiation hardened. [Ref. 1]

This method of spectral analysis is concerned with signals that contain more subtle types of periodicity that do not give rise to spectral lines, but which can be converted into spectral lines with a nonlinear time-invariant transformation of the signal. The spectral correlation density function for a discrete real-valued signal $x(n)$ is defined as:

$$S_x^{\alpha}(k) = \sum_{k=-\infty}^{\infty} R_x^{\alpha}(k)\, e^{-j2\pi fk},$$

which is the Discrete Fourier Transform of the cyclic correlation function:

$$R_x^{\alpha} = \lim_{N\to\infty} \frac{1}{2N+1} \sum_{n=-N}^{N} [x(n+k)\, e^{-j\pi\alpha(n+k)}]\, [x(n)\, e^{j\pi\alpha n}]^{*}$$

where α is the cyclic frequency.

A particularly useful application of cyclic spectral analysis is the investigation of modulation techniques, especially spread spectrum. Figure 1 [Ref. 2 p. 28] is a plot of the cyclic spectrum of a bipolar phase-shift keyed (BPSK) signal. The magnitude of the cyclic spectrum is plotted as the height above the α - f plane, where f is the spectral frequency and α is the cyclic frequency. The power spectral density function is represented on the $\alpha = 0$ line.

The computational complexity of cyclic spectrum analysis, which far exceeds that of conventional spectrum analysis, limits its use as a signal and systems analysis tool. The operations involved in the algorithms are common to most signal processing algorithms: Fourier transformations, convolution, and product modulations [Ref. 1]. In this application, the high number of operations required are too great for general purpose computers. Computing the cyclic spectrum algorithms can best be accomplished by using Application Specific Integrated Circuit Design (ASIC) in Very Large Scale Integrated Circuits (VLSI).

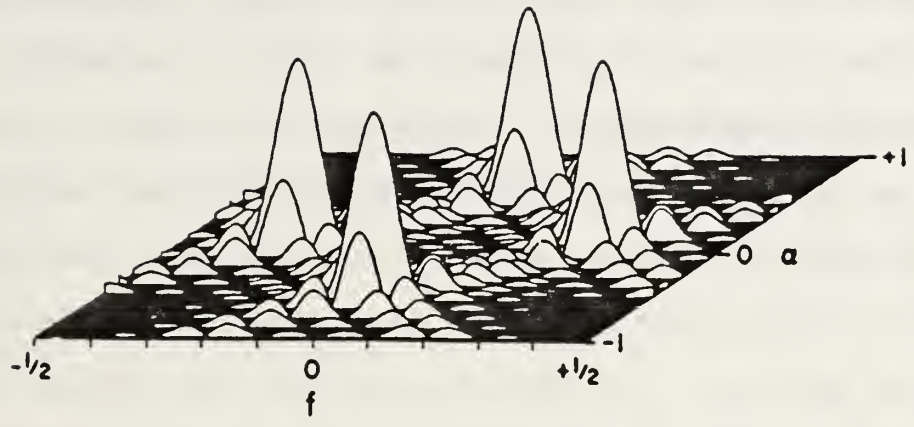


Figure 1. Cyclic Spectrum for a BPSK Signal

B. GENESIL SILICON COMPILER (GSC)

One method of ASIC design is silicon compilation. A silicon compiler is an automatic translation tool that converts a behavioral description to mask level description. In other words, a silicon compiler allows an engineer who is not expert in IC design, to design an IC. Because of low design costs, silicon compilation is also ideal to implement an IC design that will not have a large production quantity. A major problem with silicon compilers is low component density which translates to large silicon area and slower clock speed. To alleviate this problem, new versions of silicon compilers are providing more capability in automatic floorplanning and routing.

The GSC provides the user with the capability of designing VLSI circuits from high level system description to manufacture tapeout by producing the IC circuits from architectural descriptions. Huber [Ref. 3:p. 88] states that there are two significant limitations to the GSC Version 7.1 which he used: component density and vertical feedthrough. The most significant is the inability to achieve high component density. In Huber's parallel multiplier design [Ref. 3:pp. 86-88], an attempt was made to establish vertical feedthrough between adjacent multiplier levels with no success. Since that time GSC Version 8.0 and the Logic-Compiler (AutoLogic) have been installed. This software offers more capabilities to overcome these limitations. The Logic-Compiler performs synthesis and optimization on an input netlist representation of a design to produce an output design optimized for area and performance. Appendix A gives a more complete description of Genesil 8.0 and the Logic-Compiler.

C. THESIS GOALS

The motivation for this thesis is to implement a cyclic spectrum analyzer (CSA) using ASIC VLSI design. The fundamental building blocks for the CSA are the floating point multiplier, adder, and the Fast Fourier Transform (FFT) butterfly. The primary goal of this thesis is to design these processing elements: a floating point multiplier, a floating point adder, and a rate-1/4 radix-4 complex floating point FFT

butterfly using a 20-bit word that can operate at a minimum rate of 40 MHz. To achieve this goal, investigation of high speed arithmetic and the capabilities of Genesil and Logic-Compiler is required. Chapter 2 presents an indepth investigation of high speed arithmetic design.

II. HIGH SPEED DIGITAL ARITHMETIC

A. NUMBER SYSTEMS

1. Introduction

Representation of numbers within a digital system is accomplished with a group of bits. The number of bits used to represent a number determines the total number of representable values. For each additional bit added to the representation, the number of representable values doubles. For example, there are 2^N representable values in a N bit binary number. What these values represent depend on the number system chosen by the designer. Integer representations include ones' complement, two's complement, and excess code. Rational number representations utilize the integer number systems to implement fixed point and floating point representations of fractional numbers.

2. Integer Number Systems

a. Unsigned

The simplest integer system is the unsigned system. The binary numbers just represent unsigned numbers. The range of representable numbers is from 0 to 2^{N-1}, where N is the number of bits in the representation. Each bit position k has associated with it a value of 2^k and the value represented by the collection of bits is described as:

$$V_{UNSIGNED} = \sum_{i=0}^{N-1} b_i \times 2^i.$$

Where b_i is the one or zero in position i. Unsigned numbers are easy to manipulate but they can only represent positive integers. [Ref. 4:pp. 31-32]

b. Two's Complement

The most common method to represent signed numbers is the two's complement number system. The range of representable numbers in a N-bit word are from -2^{N-1} to $2^{N-1} - 1$. Negative numbers are represented by subtracting the unsigned value of the number from 2^N. The value for any N bit two's complement number is given by:

$$V_{2's} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i.$$

For example, let $N = 4$ and the number to represent be -7 = -0111 in binary. The number is represented in two's complement as $2^4 - 7$ which is 1000 - 0111 = 1001 in binary. [Ref. 5:pp. 190-193]

Although the most significant bit is not defined as the sign bit, it still is considered as such. If the most significant bit is set, the value will be negative. This is

because the most significant bit carries more weight than all of the other bits added together.

The reason that two's complement is such a popular system is its circular nature. This is illustrated in Figure 2 [Ref 5:p. 192]. The primary drawback to using two's complement number system is that it requires a relatively complicated conversion from signed magnitude to two's complement or vice versa. Two's complement multiplication also requires more hardware than unsigned or magnitude multiplication.

c. Ones' Complement

Another number representation, ones' complement, requires a much simpler conversion procedure from signed magnitude. The ones' complement conversion requires only that each bit of the signed magnitude binary number be inverted for negative numbers. The ones' complement representation of a binary number N is formulated by $N_{ones'} = (2^n - 1) - N$, where n = the number of bits and N = the unsigned number to be inverted. As shown in Figure 3 [Ref. 5:p. 194], ones' complement also has a circular nature except that it has two representations of zero; 0000 and 1111. Ones' complement addition develops a special situation when a carry is generated. Since there are two zeros in the number system, the sum will be in error by one from the correct answer if a carry is generated. This is corrected by the "end-around"

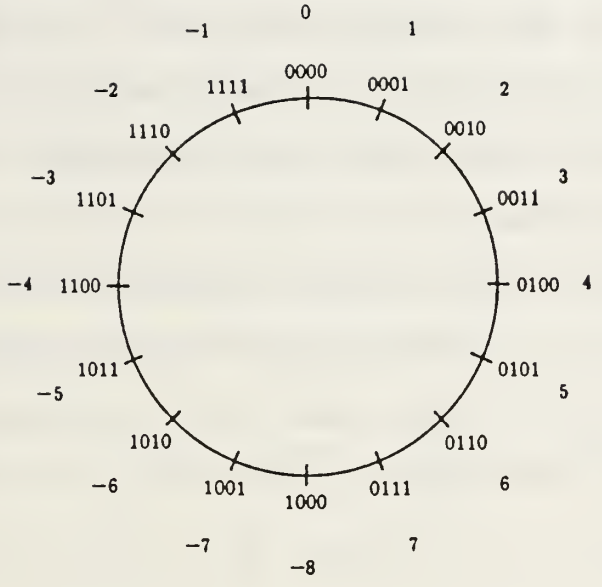


Figure 2. Two's Complement Representation

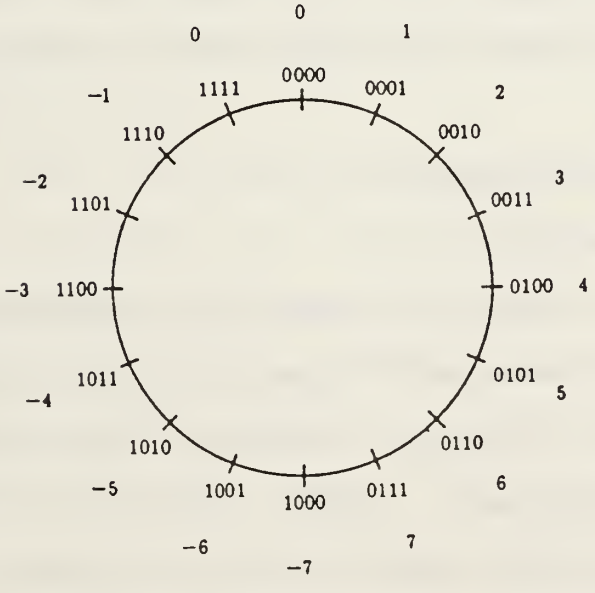


Figure 3. Ones' Complement Representation

carry. The "end-around" carry adds one to the sum if a carry is generated in the original addition. Using this number system requires as much or more hardware to implement arithmetic operations as the two's complement system.

d. Excess Code

Excess code number representation utilizes an excess number that is added to the value of the number to be represented. If V is the number to be represented and E the excess then the excess code number S is:

$$S = V + E.$$

For example, the 4-bit excess 8 code for 3 is 1011_2 and the code for -3 is 0101_2. Zero is represented by 1000 in binary. Excess code can be converted to two's complement by inverting the most significant bit of the excess code number. The most prevalent use of excess code is to store exponents in floating point numbers.

3. Rational Numbers

a. Fixed Point

Fixed point rational number representation is much like integer representation except that the radix point is not directly to the right of the least significant bit of the number. The placement of the radix point is established purely to satisfy the requirements of the user or designer for fractional or integer numbers. If the information to be represented contains fractional values, then assumption of a

radix point establishes a fixed point system that is so adjusted that it can cover the necessary range [Ref. 4:p. 36]. Addition in fixed point systems is done exactly as for integer operations. For multiplication, care must be taken to assure that the radix point is in the correct place and that the correct bits are preserved after an operation. The value of a two's complement fixed point system is:

$$V_{FIXED\ POINT} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}.$$

Where p is the position of the radix point; the number of bit positions to the left of the least significant bit where the assumed radix point is found. If $p = 0$, then the fixed point system would be the same as the integer one. This enables the designer to determine the smallest value required to meet the needs of the system and select the number system accordingly.

b. Floating Point

(1) Format. Many applications require the ability to represent information of a much greater or smaller magnitude than possible with fixed point systems. The use of scientific notation solves this problem in the decimal number system. A similar system is used to represent large and small numbers in digital arithmetic systems. This number system is called the floating point number system. This type of number system does not expand the quantity of representable values,

it modifies the way in which the values are interpreted. [Ref. 4:p. 42]

To specify a floating point number, seven different pieces of information are required: base of the system, sign, magnitude, and base of the mantissa, and the sign, magnitude, and base of the exponent. [Ref. 4:p. 42] In most cases, the base of a digital number system, the base of the mantissa, and the base of the exponent is 2. A floating point number, as described above, will have the following format:

$$(Sign)\,Mantissa \times Base^{EXPONENT}.$$

The sign bit denotes the sign of the floating point number, Usually represented as a 0 for positive and a 1 for negative. The mantissa is used to identify the significant bits of a number value. The base denotes the radix of the system, usually 2. This value is not stored in a digital system but is part of the definition of the number system. The location of the value of a floating point number on the real number line is determined by the exponent.

(2) Mantissa. The number of bits in the mantissa determines the accuracy the floating point numbers represented. The format of the mantissa usually includes a "hidden" bit when representing normalized numbers. The use of a "hidden" bit increases the number of representable mantissas by 2. To compute the range of the number system, the minimum

and maximum allowable values for the mantissa must be determined.

The minimum and maximum value of the mantissa depend on the use of a "hidden" bit and the acceptance of denormalized numbers. Normalized numbers are floating point numbers that are forced to have a 1 in the most significant bit position of the normalized mantissa. Using a "hidden" bit for the most significant bit is ideal since it will always be 1. In the IEEE standard [Ref. 6] for binary floating point numbers and in most other systems, the radix point is located to the right of this "hidden" bit. If the system allowed denormalized numbers, the "hidden" bit could be 0, thus allowing a greater range of representable numbers. For example, a 4 bit normalized mantissa with a "hidden" bit has a minimum value of 1.0000 and a maximum value of 1.1111. The same system, except that it allows denormalized numbers, has a minimum value of 0.0001 and a maximum value of 1.1111. It is possible to use any integer number system for the mantissa including the systems discussed in para. 2. The most common method is signed magnitude, which is the IEEE standard [Ref. 6] for floating point mantissas.

(3) Exponent. The exponent along with the radix of the system determines the range of the floating point system. The exponent also needs to have a sign to represent floating point values less than the smallest representable value of the

mantissa. In a normalized base 2 floating point number system, the smallest mantissa is one. To represent fractional values in this system, the exponent must be negative. For example, .5 is represented by a 1 in the mantissa and a -1 in the exponent:

$$1 \times 2^{-1} = .5 \ .$$

Like the mantissa, any integer number system would be sufficient for the number representation in the exponent, but the most commonly used method is excess code. The IEEE standard [Ref. 6] uses excess code.

Zero representation in a normalized system is done in the exponent. Usually the smallest representable value in the exponent is reserved to indicate a true zero value. This must be done because the "hidden" bit is always a one, which means the mantissa is always nonzero. In systems which allow denormalized numbers, there is a zero in the "hidden" bit when a denormalized number is represented, usually denoted by all zeros in the exponent. In this case, true zero is represented by the smallest value in the exponent and all zeros in the mantissa.

B. HIGH SPEED INTEGER ADDERS

1. Introduction

The addition function in an arithmetic computation system is the most fundamental of add, subtract, multiply, and

divide functions. All of these operations can be implemented by some combination of the add function. The full adder cell is the fundamental building block in the ripple-carry and carry-save adders. The two other high speed adders to be discussed, conditional sum and carry-lookahead, are synchronous and do not require the use of the full adder cell.

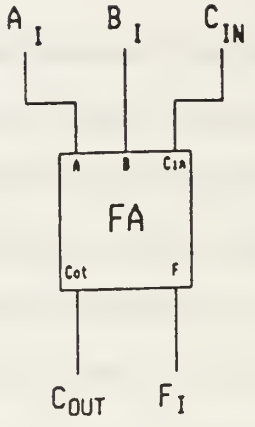
2. The Full Adder

The function of a full adder is to add two bits and the carry from the next less significant bit to produce a sum and a carry out to next more significant bit. A functional diagram is shown in Figure 4(a) [Ref. 4:p. 71]. The truth table for the function is shown in Figure 4(b). As shown, the three input bits, A_i, B_i, and C_{in}, are summed to produce two bits, F_i, the sum, which has the same significance as the input bits, and C_{out}, which is one bit more significant. Figure 4(c) show the Karnaugh maps for C_{out} and F_i with the resulting sum of products Boolean equations. These equations are implemented with random logic as shown in Figure 4(d). [Ref. 7:pp. 70-71]

3. Two Operand Adders

a. Ripple-Carry Adder

The ripple-carry adder is just a group of full adders cascaded to the width of the desired word length. The C_{out} of one bit is wired to the C_{in} of the next significant bit. This is not a high speed adder design because it requires two



(a)

A_i	B_i	C_{in}	C_{out}	F_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b)

C_{in} \ A_iB_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

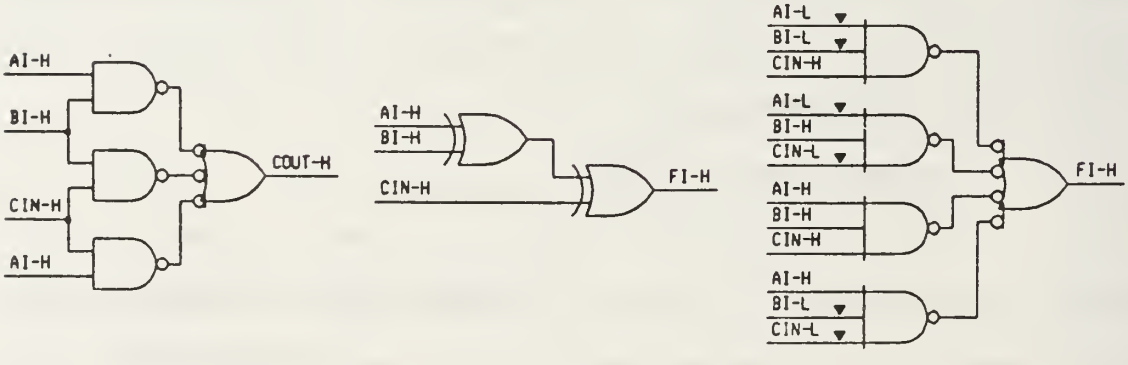
$$C_{out} = A_iB_i + B_iC_{in} + A_iC_{in}$$

C_{in} \ A_iB_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$F_i = \bar{A}_i\bar{B}_iC_{in} + \bar{A}_iB_i\bar{C}_{in} + A_iB_iC_{in} + A_i\bar{B}_i\bar{C}_{in}$$

$$F_i = A_i \oplus B_i \oplus C_{in}$$

(c)



(d)

Figure 4. Full Adder Cell Design

gate delays for every bit in the width of the numbers to be added. For example, in the 8 bit adder shown in Figure 5, the delay to final C_{out} is 16 gate delays. This adder can be made synchronous by the insertion of appropriately placed flip-flops.

b. Conditional Sum Adder

In the case of ripple-carry adder, the carry must propagate through the length of the word. This is an unacceptable delay for high speed arithmetic operations. The conditional sum adder overcomes this problem by generating distant carriers and using these carriers to select the true sum outputs from two simultaneously generated conditional sums under different carry input conditions. [Ref. 7:p. 78] Conditional sum adders offer significant speed gains over ripple-carry adders by utilizing logic gates and multiplexers with small fan in and fan out. The delay is proportional to $\log_2 N$ instead of N as in the ripple-carry case. The major disadvantage is a large increase in area required for hardware.

A 7-bit two-operand adder using the conditional sum algorithm is illustrated in Figure 6 [Ref. 7:p. 79]. For the example, the inputs are $A = 1101101_2$ and $B = 0110110_2$ with no external carry in. S_i's are the conditional sums within the adder. Subscripts indicate bit position and superscripts indicate the assumption of a carry or no carry into the lowest

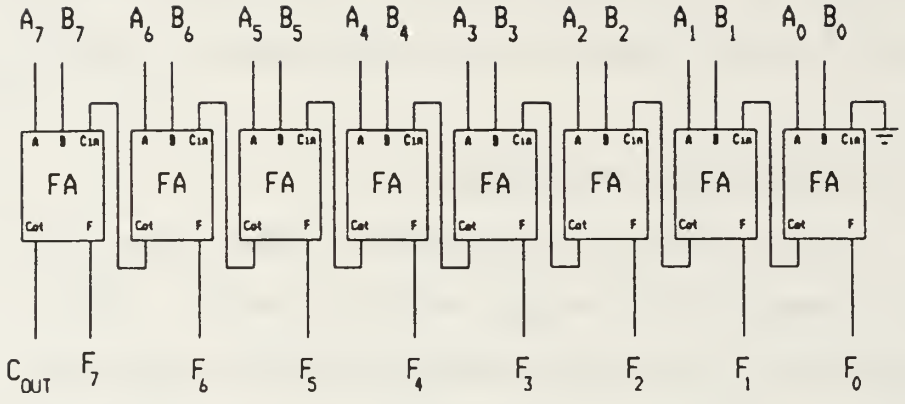


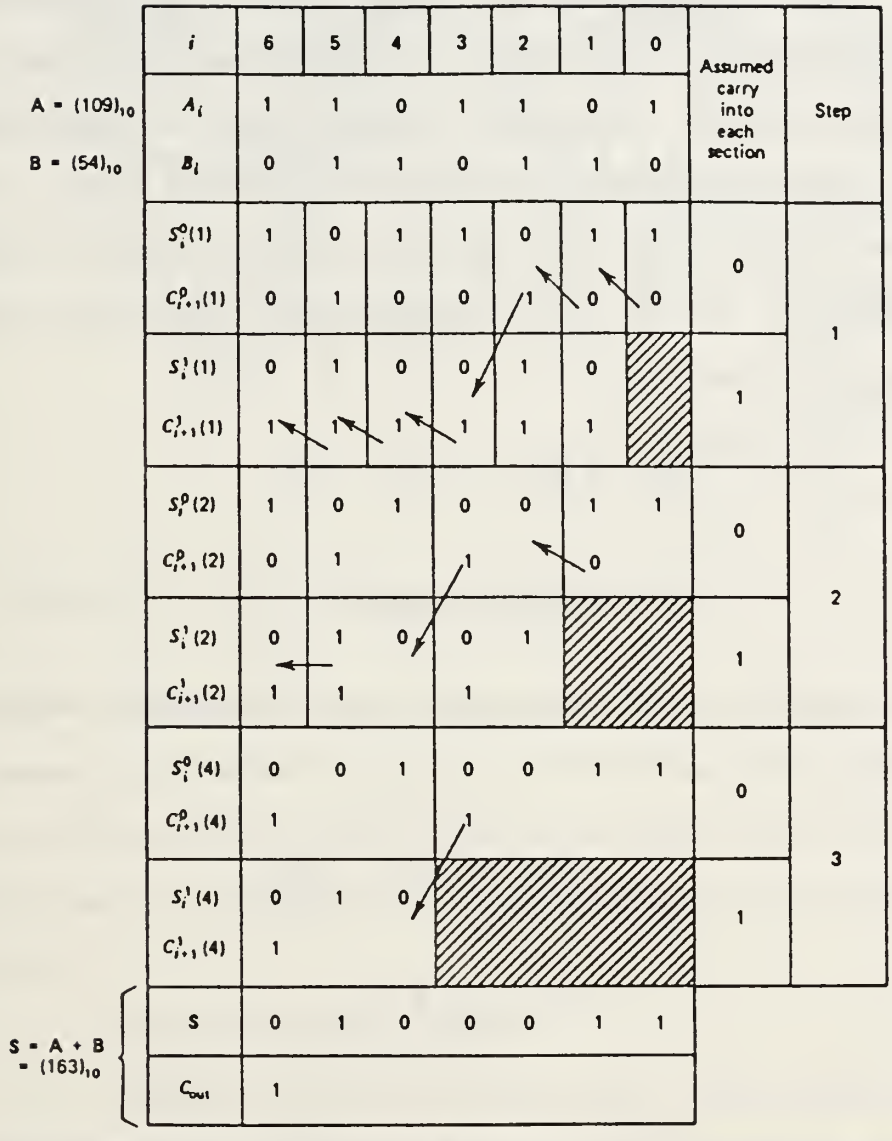
Figure 5. 8-bit Ripple Carry Adder

order bit position of a section. There are $\lceil n/k \rceil$ sections in $S_0(k)$ or $S_1(k)$ for an n-bit addition. The number of steps (t) required is given by:

$$t = \lceil \log_2 n \rceil.$$

Where n is the number of bits in the adder.

The adder represented in Figure 6 has $n = 7$. Therefore, $t = \lceil \log_2 7 \rceil = 3$ steps are required to complete the addition. Step one has a carry and a no carry into each bit position, so there are 7 sections. The section size doubles for each successive step with a carry and no carry into each least significant bit position of each section. The arrows in Figure 6 show, for the example inputs, how the carries are



Note: The arrows show the actual carries generated between sections. The initial carry to the rightmost section is always assumed zero.

Figure 6. 7-bit Conditional Sum Adder Algorithm

generated between sections and how they are eventually used to select the true sum and carry outputs. [Ref. 7:pp. 78-80]

The conditional sum adder can be implemented with 2-input multiplexers, and gates, or gates, and inverters. The initial conditional sums and carries can be generated in parallel using the random logic gates indicated above. If the conditional sum and carry of ith bit position with no carry in are denoted by S_i^0 and $C_{i+1}{}^0$ respectively, then the Boolean equations for these are:

$$S_i^1 = A_i \oplus B_i = A_i \overline{B}_i + \overline{A}_i B_i;$$

$$C_{i+1}^0 = A_i B_i.$$

Similarly for the conditional sum and carry with a carry in (S_i^1 and $C_{i+1}{}^1$):

$$S_i^1 = A_i \odot B_i = A_i B_i + \overline{A}_i \overline{B}_i;$$

$$C_{i+1}^1 = A_i + B_i.$$

Each input bit position must generate these sums and carries. The carries will eventually be used to select the final sum and carry out. The Conditional Cell (CC) in Figure 7 [Ref. 7:p. 81] generates these conditional sums and carries. Figure 7 also illustrates the hardware required to implement the 7-

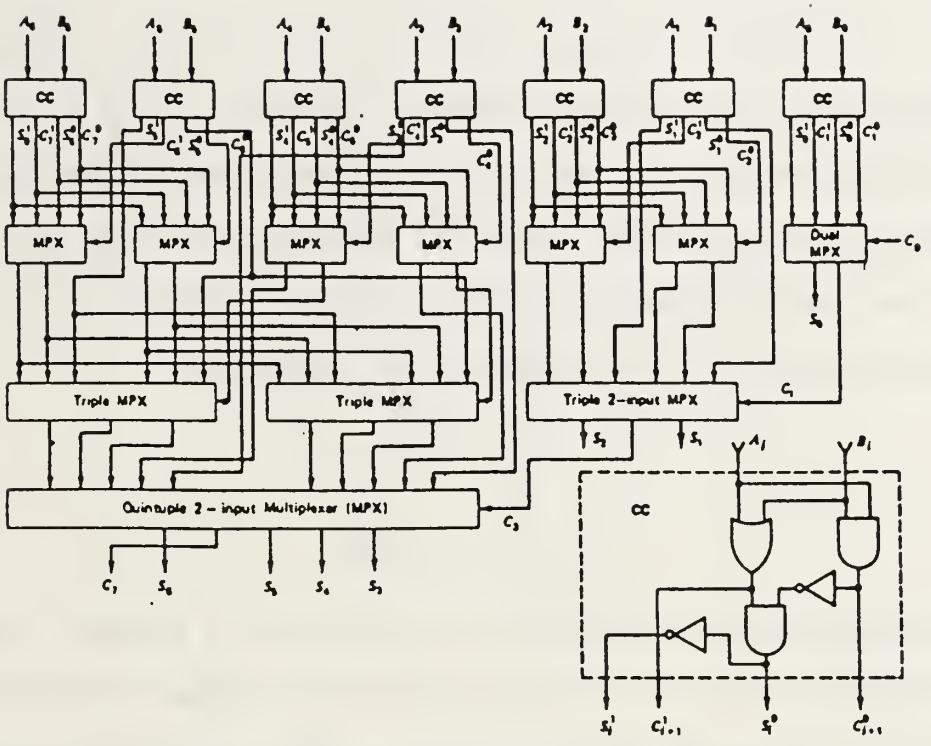


Figure 7. 7-bit Conditional Sum Adder

bit adder described in Figure 6. The first stage of multiplexers selects the 0th bit of the final sum. The second level selects the 1st and 2nd bits and the final multiplexer outputs the 3rd through the 6th bits and the carry out of the final sum.

c. Carry-Lookahead Adder

Another adder that overcomes the carry propagate problem is the carry-lookahead adder. Carry ripple is eliminated by using additional random logic to simultaneously generate the carries entering all of the bit positions. This results in a constant add time regardless of the length of the adder.

Let $A = A_{n-1},\ldots A_1,A_0$ and $B = B_{n-1},\ldots B_1,B_0$ be the inputs to a n-bit carry-lookahead adder. S_i and C_i are the sum and carry outputs of the ith bit position of the adder. Two functions must be generated for each bit position to implement the carry-lookahead algorithm; carry generate (G_i) and carry propagate (P_i). The Boolean equations are:

$$G_i = A_i \cdot B_i;$$

$$P_i = A_i \oplus B_i.$$

The ith carry generate function produces a binary 1 if a carry is generated at the ith bit position independent of the less significant sums and carries. The ith carry propagate function produces a 1 if a carry is generated by a carry in from the less significant bits. Although the obvious implementation for generating the P_i's and G_i's would be to use AND gates and exclusive OR gates, a NAND gate implementation is possible and probably more economical in area usage. Figure 8 illustrates the NAND gate implementation of a n-bit wide carry generate and carry propagate unit for a carry-lookahead adder. The following relations result after substituting P_i and G_i into the sum and carry equations for a full adder cell:

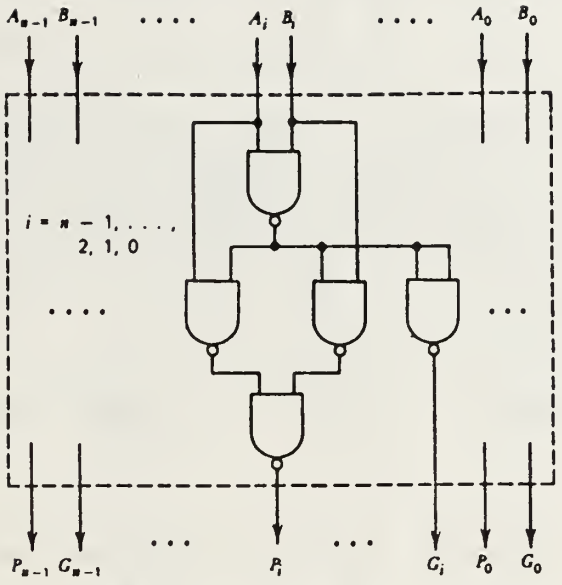


Figure 8. *n*-bit Carry Generate/Propagate Unit

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

$$= P_i \oplus C_{i-1};$$

$$C_i = A_i \cdot B_i + (A_i \oplus B_i) \cdot C_{i-1}$$

$$= G_i + P_i \cdot C_{i-1}.$$

These equations show that the sum and carry of every bit position is dependent only on the simultaneously generated P_i's and G_i's and the carry in from the next less significant bit position. To make this adder truly parallel and high speed these carries must be generated in parallel also. To

accomplish this the equation for C_i can be used recursively as follows:

$$C_0 = G_0 + C_{-1}P_0;$$

$$C_1 = G_1 + C_0P_1$$

$$= G_1 + G_0P_1 + C_{-1}P_0P_1;$$

$$C_{n-1} = G_{n-1} + G_{n-2}P_{n-1} + \cdots + C_{-1}P_0P_1 \cdots P_{n-1}.$$

Where C_{-1} is the external carry in of the adder. These equations can be realized with random logic. Figure 9 [Ref. 7:p. 86] is the logic circuit diagram of a 4-bit carry lookahead unit. Obviously, the size of the carry lookahead unit is limited by the fan-in of the random logic being used.

The final sum is generated with an array of XOR gates as shown in Figure 10 [Ref. 7:p. 85] which is called the summation unit. Figure 11 [Ref. 7:p. 89] illustrates how the carry generate/propagate unit, the carry lookahead unit, and the summation unit are combined together to form an 8-bit carry lookahead adder.

Fan-in limitations of the random logic used to build a carry lookahead unit is a severe constraint that must be solved. It can be solved by the using the block carry lookahead unit. This unit generates a block propagate (P^*) if a carry into the block would force a carry out of the block.

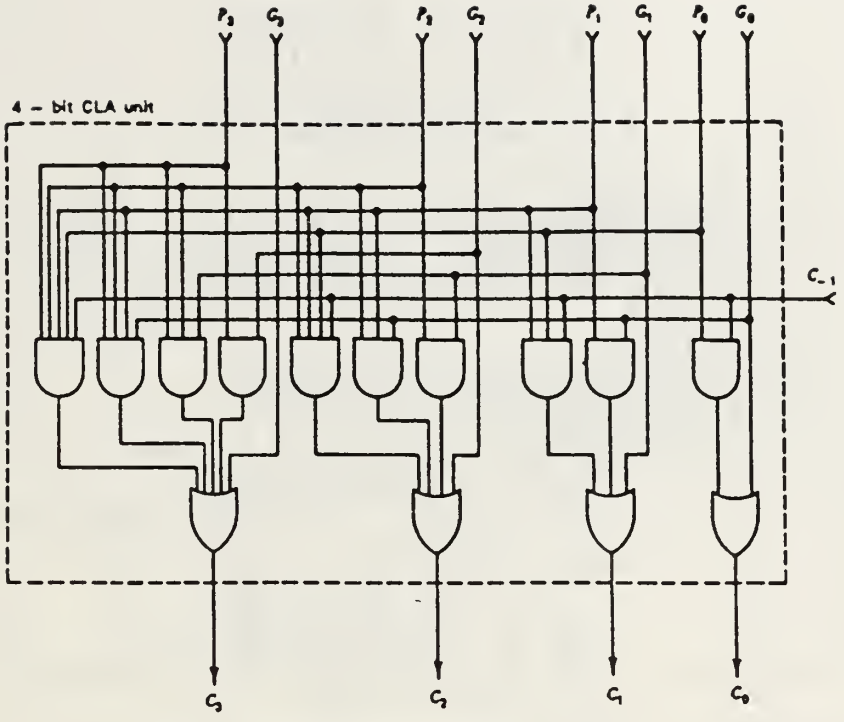


Figure 9. 4-bit Carry Lookahead Unit

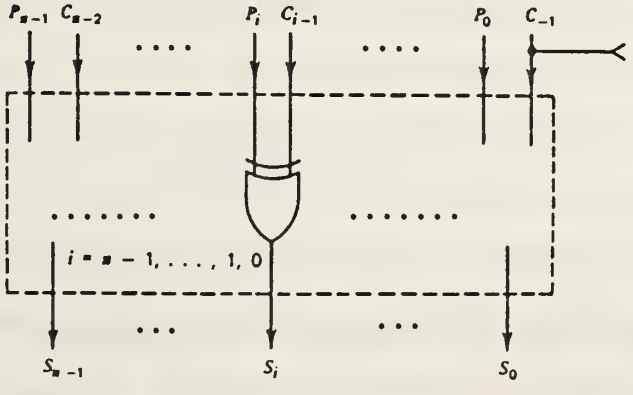


Figure 10. Summation Unit

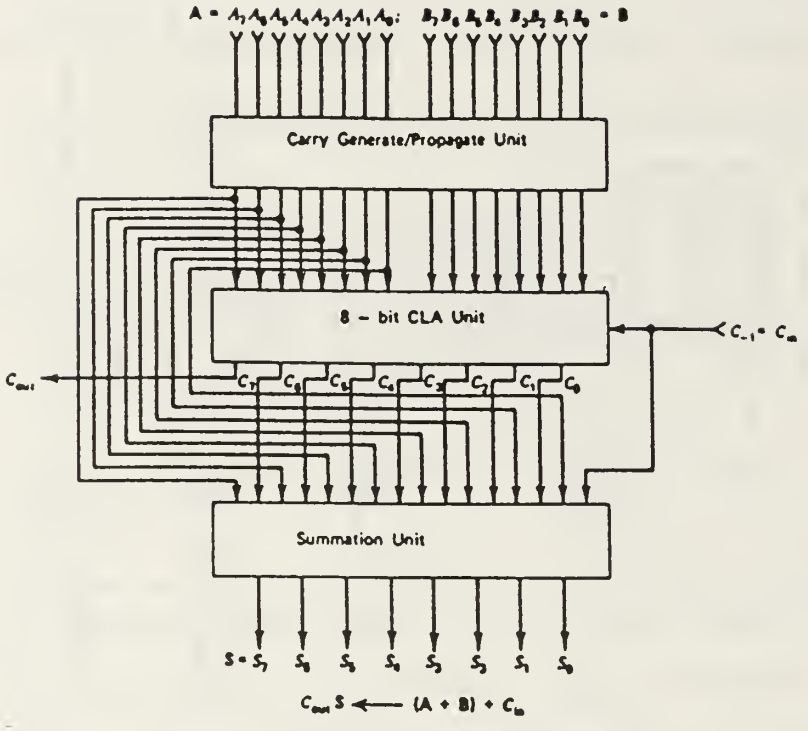


Figure 11. 8-bit Carry Lookahead Adder

The block also generates a block generate (G^*) if there is a carry that originated within the block. [Ref. 7:pp. 87-88] Figure 12 [Ref. 7:p. 87] is logic circuit diagram of a 4-bit block carry lookahead unit. The equations for P^* and G^* are as follows:

$$P^* = P_0P_1P_2P_3;$$

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3.$$

These blocks can be combined together to create a carry lookahead adder of any size. Figure 13 [Ref. 7:p. 90] is a 32-bit carry lookahead adder using 4-bit and 8-bit block carry lookahead units.

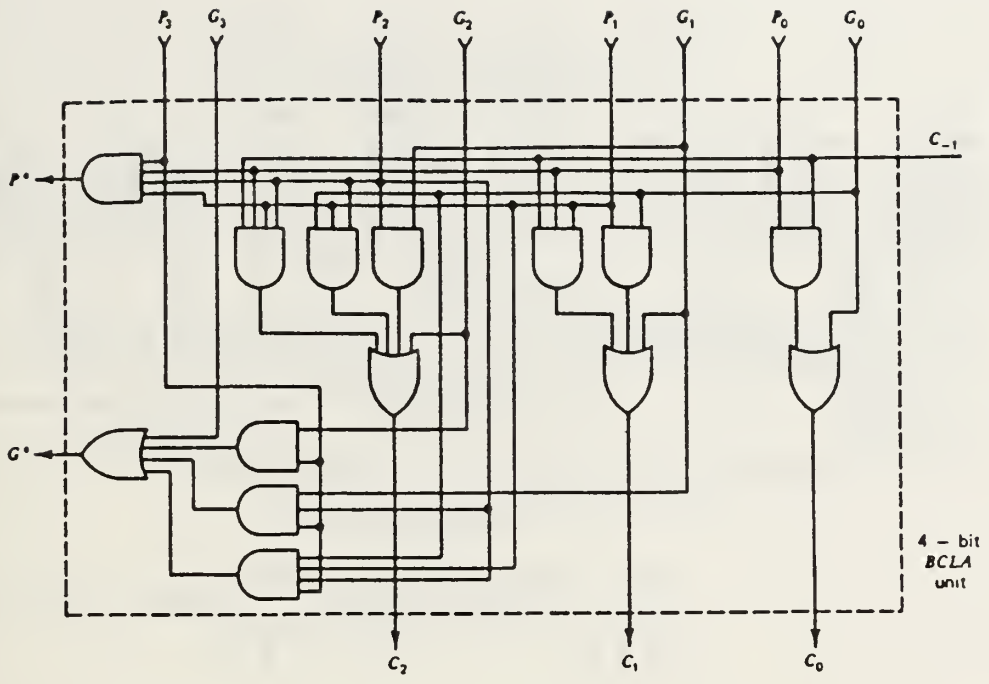


Figure 12. 4-bit Block Carry Lookahead Unit

4. Multioperand Adders

a. Introduction

If there is a requirement to add more than two numbers together, such as summing partial products in a multiplier, then the number of two operand adders must increase. For N inputs there must be N-1 two operand adders. For large number N, the adder will be intolerably slow and large. The solution is to build multioperand adders from random logic. The carry-save adder is one example of a multioperand adder.

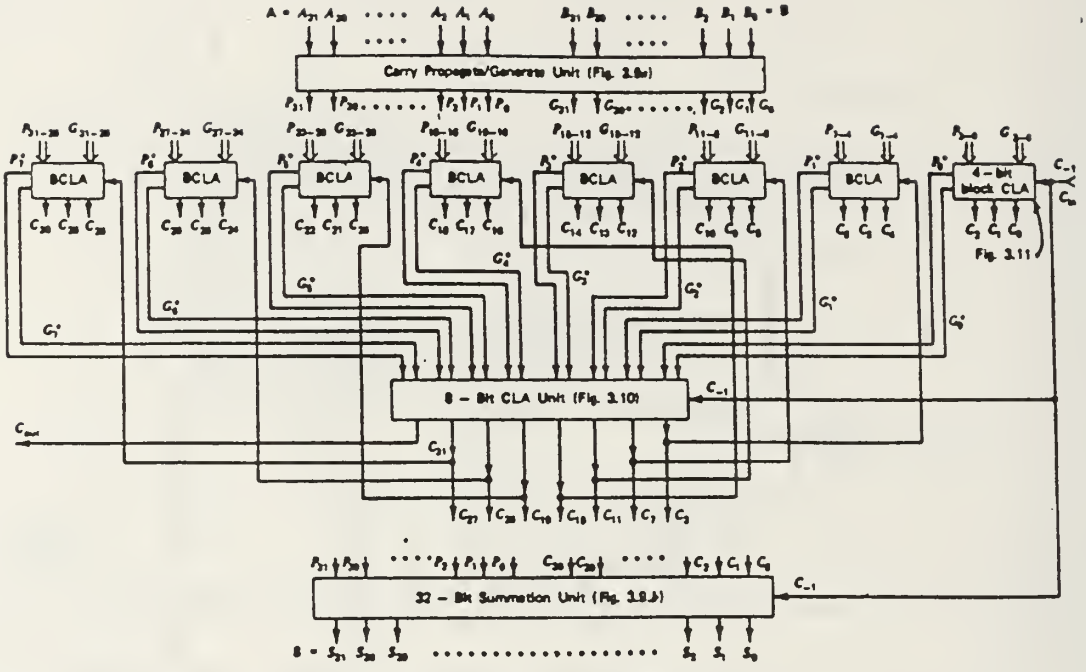


Figure 13. 32-bit Carry Lookahead Adder

b. Carry-Save Adder

The carry-save adder is constructed from full adder cells like the carry ripple. The difference is only that in the carry-save adder the carry out from each cell is not propagated to the carry in of the next significant cell. This carry out is saved for the next level of adders. This leaves three inputs into the adder cell of equal precedence. The cell produces one output of the same significance and one output of one bit greater significance. Utilizing the adder cell in this way is called row reduction. The full adder is a 3-to-2 row reduction unit. Figure 14 [Ref. 6:p. 101] illustrates

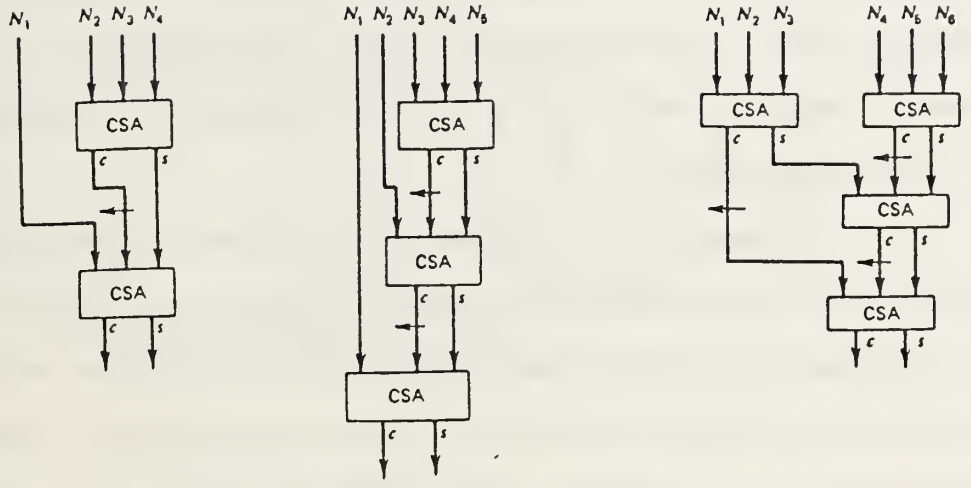


Figure 14. Carry Save Adder Trees

how carry-save adders (CSA) can be configured to produce row reduction units with a various number of inputs. Since the carry-save adder does not solve the problem of or complete the carry ripple, a two operand adder must be used in the final stage of adders to complete the sum.

C. HIGH SPEED INTEGER MULTIPLIERS

1. Standard Multipliers

a. Introduction

An integer multiply in the binary number system is much like that done in the decimal number system. This procedure is illustrated in Figure 15 [Ref. 4:p. 83]. PP_0 - PP_4 are called the partial product rows. Each partial product

is generated by a binary multiply: the and function. The final product is the sum of the properly aligned partial product rows. This indicates the requirement of a binary product module (AND) and a summing module to complete the multiply function.

Standard multipliers are based on the add-shift method for multiplication. Multipliers of this type include the standard add-shift, multiple shift, multiple shift with overlapped scanning, and the Booth multiplier. The Booth multiplier, a variant of the add-shift method, uses string recoding.

b. Standard Add-Shift Multiplier

The simplest method for doing the multiply is the standard add-shift method. Figure 16 [Ref. 4:p. 84] illustrates one implementation of this method using standard integrated circuits (IC). This 8 x 8 multiplier has 2 8-bit inputs and 1 16-bit output. The multiplier is initially loaded into the shift register to provide one bit into the and gates with all of the multiplicand to generate the first row of partial products. This row is then added with the sum in the output register (initially zero). This is called the accumulation sum. An 8-bit adder is used because the partial product addition is done from the least significant partial product to the most significant partial product. The output

					A_4	A_3	A_2	A_1	A_0
				×	B_4	B_3	B_2	B_1	B_0
$PP_0 \rightarrow$					$A_4 \cdot B_0$	$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
$PP_1 \rightarrow$				$A_4 \cdot B_1$	$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
$PP_2 \rightarrow$			$A_4 \cdot B_2$	$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		
$PP_3 \rightarrow$		$A_4 \cdot B_3$	$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$			
$PP_4 \rightarrow$	$A_4 \cdot B_4$	$A_3 \cdot B_4$	$A_2 \cdot B_4$	$A_1 \cdot B_4$	$A_0 \cdot B_4$				
$PR \rightarrow$	Sum of all rows								

Figure 15. 5-bit Multiply

of the adder is then put into the 9 most significant bits of the output registers. For the next iteration, the multiplier is shifted one right and outputs the next significant bit is input to the AND gates with the multiplicand. The result from the previous adder iteration is shifted by hard wiring the accumulation sum to shift one bit to the right every iteration. This shift is necessary to line up the accumulation sum with the appropriate bit positions in the partial product. To complete the multiply, 8 iterations (8 PROD-CLK cycles) must be done. For a N x N bit multiply there must be N iterations. This is much too slow for high speed multiplication.

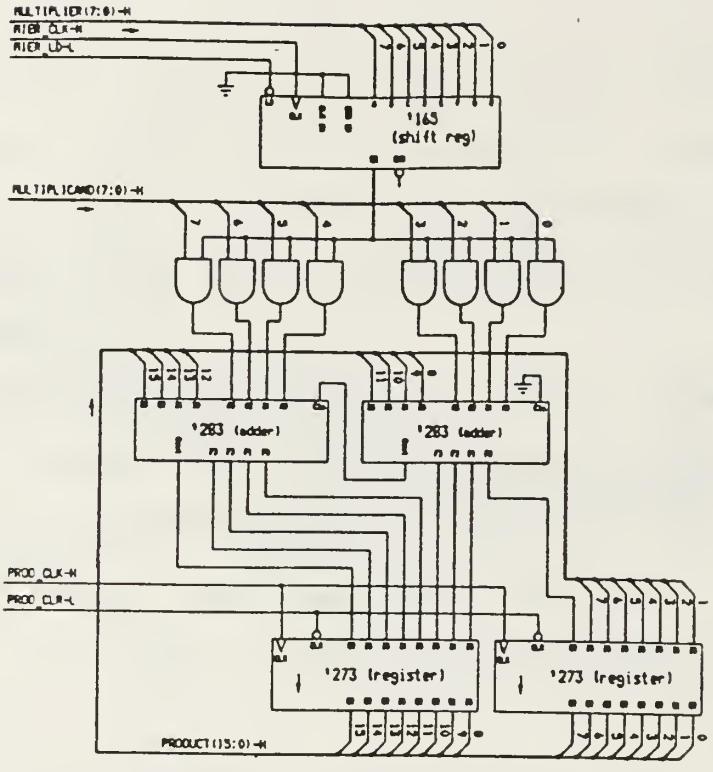


Figure 16. 8 x 8 Multiplier

c. Multiple-Shift Multiplier

The slowness of the add-shift method can be alleviated by using more than one multiplier bit per cycle. To accomplish this requires multiple bit scanning and multiple shifts after each addition. For example, the total number of add-shift cycles can be reduced by half, if two multiplier bits are examined at a time. The hardware required is greater than the one bit scanning method.

When scanning two bits at a time there are four possible actions instead of just add the multiplicand or add zero. This decision was made with AND gates. Table 1 shows the four situations with the correct values to added to the

partial product. The *A* represents the multiplicand. To generate 2*A* the multiplicand *A* is asynchronously shifted one bit position to the right. The decision to add 2*A* and/or *A* is also done with AND gates. Figure 17 [Ref. 7:p. 142] illustrates the configuration for two bit scanning. Since there will be 3 operands per add: 2*A*, *A*, and the previous partial product, a carry-save adder is utilized. The carry propagate adder (carry-ripple) in Figure 17 can be replaced with a faster two operand adder to increase cycle frequency.

This multiplier is much the same as the standard add shift multiplier except that it requires shifting of more than one bit and a multioperand adder. The cycles per multiply is greatly reduced with a small increase in cycle period. As the scan width increases, the required clock cycles decrease. But the hardware complexity and the cycle period will increase as scan width increases.

d. Multiple Shift Multiplier with overlapped scanning

In the nonoverlapped bit scanning method each multiplier bit generates one multiple of the multiplicand to be added to the partial product [Ref. 7:p. 143]. When the scan width gets large then the number of multiplicand multiples to be added gets large which decreases cycle frequency. The overlapped scanning method attempts to reduce the number of multiples to be added there fore reducing adder

Table I. Multiplicand Multiples to be added to the Partial Product after Scanning 2 Multiplier Bits

Multiplier Bit 1	Multiplier Bit 0	Multiples to be Added
0	0	0
0	1	A
1	0	2A
1	1	A+2A

complexity and cycle period. The number of multiplicand multiples can be reduced by half using the overlapped scanning method over the standard multiple-shift method.

The basis of this method is that execution time can be reduced (cycle period) by shifting across a string of zeros in the multiplier [Ref. 7:p 143]. The following describes a string of k consecutive 1's in the multiplier:

Column Position → ..., $i+k$, $i+k-1$, $i+k-2$, ..., i, $i-1$, ...;

Bit Content → ..., 0 , 1 , 1 , ..., 1, 0 , ...

By the string property:

$$2^{i+k} - 2^{i} = 2^{i+k-1} + 2^{i+k-2} + \cdots 2^{i+1} + 2^{i};$$

the k consecutive ones can be replaced by the following string:

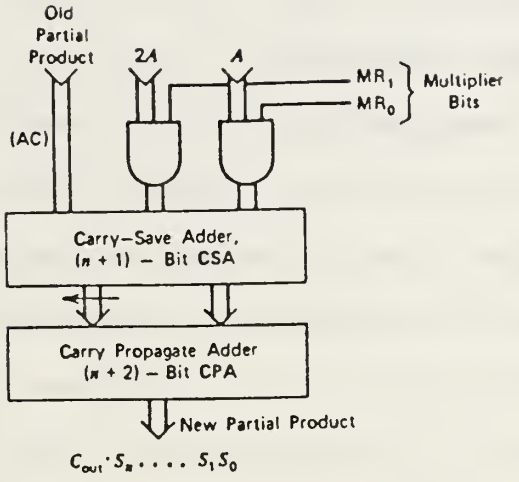


Figure 17. Adder Unit for Two Bit Scanning Multiplier

Column Positon → ..., $i+k+1$, $i+k$, $i+k-1$, ..., $i+1$, i, $i-1$, .

Bit Content → ..., 0 , 1 , 0 , ..., 0 , $\overline{1}$, 0 , ...

[Ref. 7:p. 143]

The string states that the string of k ones can be replaced by a 1 in the next more significant bit position subtracted by a 1 in the next less significant bit from the string. The 1 overbar signifies this subtraction. This is essentially replacing k consecutive adds with one add at the beginning and one add at the end of the string [Ref. 7:p. 143]. For long strings of ones this is a considerable saving. Implementing this method in a 2-bit scan width + the overlap

bit will require that addition and subtraction is possible during each cycle.

e. Booth's Multiplier

The Booth multiplier is a recoding algorithm that is also based on the string property. This method is similar to the overlapped scanning algorithm except it is used for two's complement multiplication. Let $B = B_4,B_3,B_2,B_1,B_0$, so the value of B is:

$$B_{2's} = B_4 \times (-16) + B_3 \times 8 + B_2 \times 4 + B_1 \times 2 + B_0 \times 1.$$

The above equation can be manipulated into:

$$B_{2's} = -16\times(B_4-B_3) -8\times(B_3-B_2) -4\times(B_2-B_1) -2\times(B_1-B_0) -B_0$$

The values in the parentheses in the last equation can have the values 1, 0 ,-1. The shift algorithm for this multiplication is exactly the same as the 2 bit multiple-shift multiplier. The difference is that the possible actions to take are add, subtract, or do nothing as opposed to add or do nothing in the multiple-shift multiplier. [Ref. 4:pp. 90-91]

f. Summary

There are many multiplier designs that would fit in the "shift and add" standard category that have not been discussed in this section. All of these designs can be optimized to increase speed and decrease the number of cycles. But if they are characterized as standard then they will be recursive. This implies multicycle completion and the

multiplier is not easily pipelined for high speed operations. These multipliers cannot or at least should not be used for high speed digital arithmetic. The next section discusses multiplier designs that are more appropriate for high speed arithmetic.

2. Cellular Array Multipliers

a. Standard Parallel Multiplier

(1) Introduction. The parallel multiplier is based on the observation that partial products in the multiplier can be computed in parallel [Ref. 8: p. 344]. The partial products in the standard add-shift multiplier generated its partial products with AND gates one row per clock cycle. For a N x N multiply there are N^2 partial products. To accomplish this there must be N^2 AND gates. To sum all of the partial products the multiplier requires $N(N-2)$ full adder cells and N half adder cells. Figure 15 illustrates the partial products to be summed together for a 5 x 5 multiply. Since the partial products are generated in parallel the primary delay in the computation is due to adding the partial products to get the final product.

(2) Parallel Multiplier Cell. Figure 18 [Ref. 8:p. 345] is an illustration of the parallel multiplier cell. It consists of an AND gate and a full adder. This cell is the only part that is required to build a parallel multiplier.

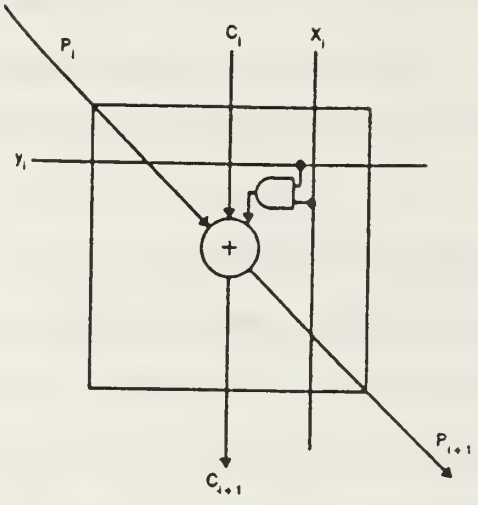


Figure 18. Parallel Multiplier Cell

(3) Parallel Multiplier. The parallel multiplier is an array of parallel multiplier cells arranged to output the unsigned product of two unsigned numbers. Figure 19(a) [Ref. 8:p. 345] is the multiplier with the partial products on each cell. Figure 19(b) has the same arrangement as in Figure 19(a) but in a square array. The latter arrangement is more convenient in VLSI to implement in hardware. It also lends itself to pipelining the multiplier into stages to increase clock frequency.

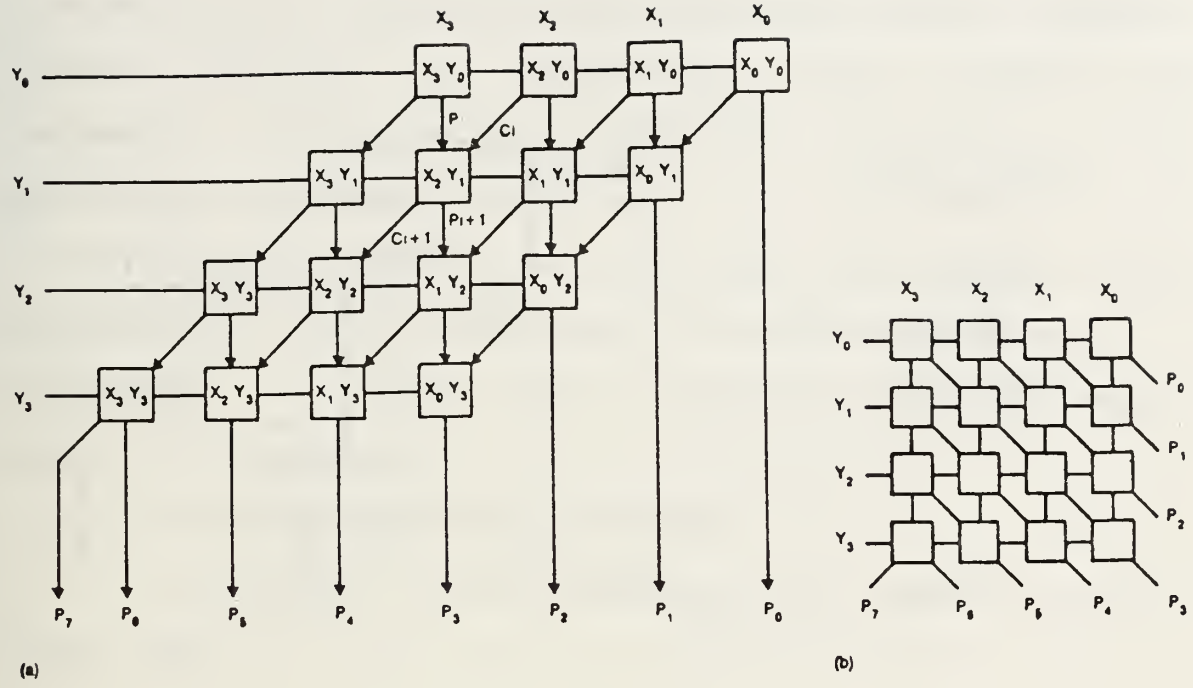


Figure 19. 4 x 4 Parallel Multiplier

b. Wallace Tree

The Wallace tree is a solution for reducing the delays due to summing the partial products in a multiplier. It takes inputs of the same significance and outputs the sum of these inputs. For example, the full adder cell is a 3 input, two output Wallace tree. Any size Wallace tree can be built from the 3 to 2 Wallace tree. Figure 20 [Ref. 7:p. 166] illustrates the full adder cell as a 3 to 2 Wallace tree and a 7 to 3 Wallace tree built from full adder cells. The Wallace tree is nothing more than a multioperand bit-slice adder.

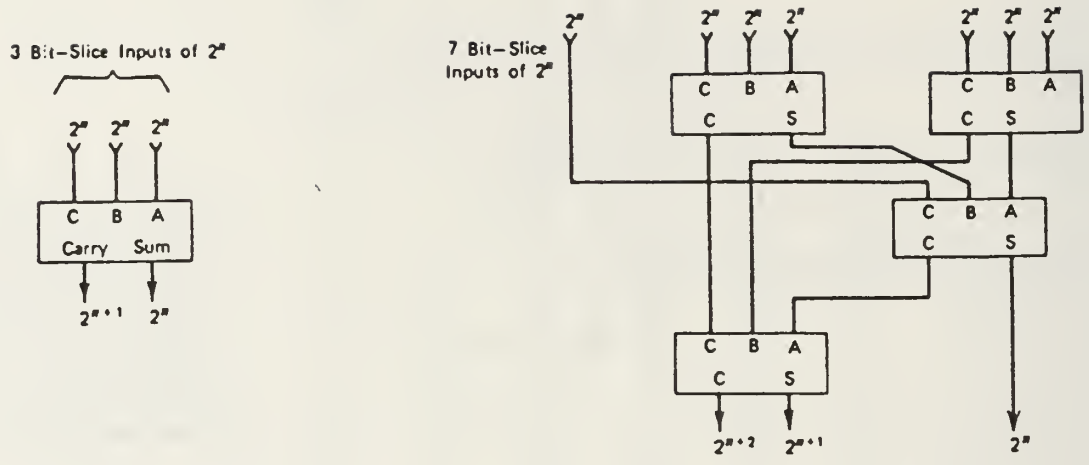


Figure 20. Wallace Trees

c. Summary

There are many other array multipliers in use but their goal is much the same, reduce the partial product adding delay. Regardless of method, the tradeoff is coldly clear, if the design must be fast then the hardware complexity must be high. More hardware translates to higher cost. Although the cellular array multipliers are faster, they are also much more expensive than the serial add-shift multipliers.

D. FLOATING POINT ARITHMETIC

1. Introduction

Using a floating point number system makes arithmetic operations much more complex. For the most part the

discussion is for operations with normalized numbers, although design differences for denormalized number systems are discussed. Multiplication is addressed first because it is much simpler than addition.

Other assumptions to made about the floating point number system are: 1) Mantissa is in signed magnitude; 2) Exponent is in excess code; 3) The floating point number system is in base 2.

2. Floating Point Multiplication

The product of two floating point numbers *A* and *B* looks like:

$$A \times B = M_A \times 2^{E_A} \times M_B \times 2^{E_B}$$

$$= (M_A \times M_B) \times 2^{E_A + E_B}.$$

The product of two floating point numbers is represented by the integer product of the mantissas times 2 raised to the sum of the exponents' power. The output sign bit is just the exclusive or of the input sign bits. Figure 21 illustrates the operations indicated in the previous equations.

The Exponent Add block is not just a simple integer adder. Since the exponents are in excess code, some additional random logic must be used. This module must also indicate if there is an overflow or an underflow generated by the add. The Mantissa Multiply block is an integer multiplier

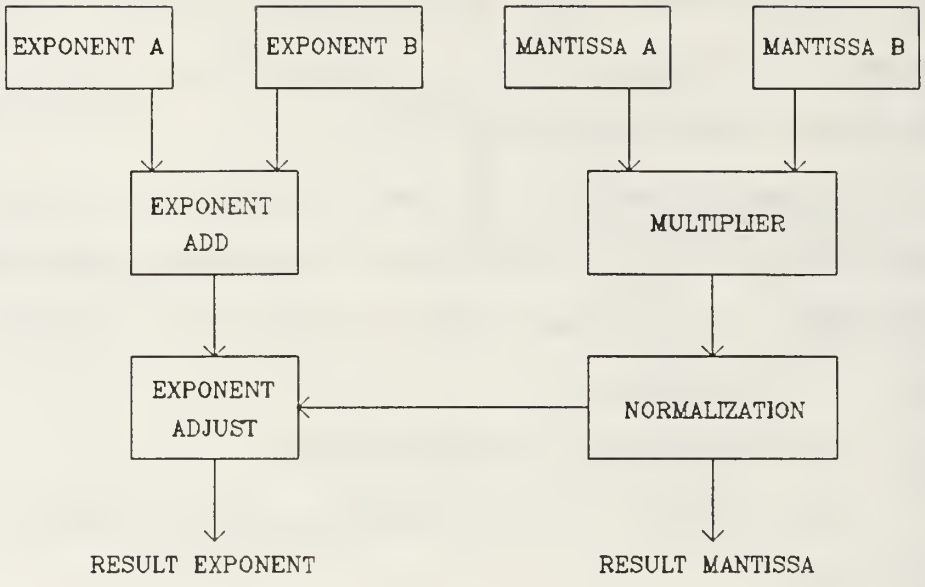


Figure 21. Block Diagram of a Floating Point Multiplier

module. This block incurs the most delay and uses the most logic of the floating point multiplier, so this block must be optimally designed for speed and hardware. No special circuitry is required because the mantissa in this system is represented in signed magnitude. The Normalization block can be broken up into 3 sub-blocks: Normalizer, Rounder, and Postnormalizer.

The Normalizer sub-block detects if the product from the Mantissa Multiply block is a normalized number and normalizes it if it is not. Since the multiplier only computes products of normalized numbers the product will be between 1 and 4. This is illustrated by a 4 bit mantissa

multiply with a "hidden" bit of the minimum and maximum values that can be represented:

Minimum: 1.0000 × 1.0000 = 1.00000000;

Maximum: 1.1111 × 1.1111 = 11.11000001.

As the mantissa length increases, the maximum product gets closer to 4. This makes the decision to normalize and the actual normalization very simple. If there is a 1 in the next most significant bit from the "hidden" bit then the mantissa must be normalized. To be normalized, the mantissa merely needs to be shifted to the right one bit and the exponent incremented by one. If the multiplier were to allow denormalized numbers, then the product would not necessarily be between 1 and 4. To normalize such a number will require a significantly larger amount of hardware to detect and shift the most significant 1 from anywhere in the product mantissa.

The Rounder sub-block rounds the product from the Normalizer to the correct number of significant bits for the number system. There are many methods for rounding: truncation, rounding, unbiased rounding, and jamming to name a few. Regardless of the method that is used, when a decision is made to round up, a 1 is added to the least significant bit of the mantissa. This add could cause carry out which will require another normalization process called the Postnormalizer sub-block.

The Postnormalizer sub-block provides normalization of the sum generated by the Rounder sub-block. The function is exactly the same as the Normalizer sub-block.

The Exponent Adjust block is merely an adder for the incrementing of the exponent generated by the Normalizer and Postnormalizer sub-block. Implementing the multiplier with denormalized numbers will require that the adder be capable of adding more than |1| for the Normalizer increment because the mantissa product may be less than 1.

3. Floating Point Addition

The primary reason that floating point addition is more difficult is that the mantissas usually have different significance. Therefore, before doing any arithmetic, they must be aligned to perform the mantissa addition. Alignment means that the exponents must be equal to correctly add the mantissas. The sum of two positive floating point numbers is:

$$A + B = M_A \times 2^{E_A} + M_B \times 2^{E_B}.$$

Assuming that $E_A < E_B$ then $E_A - E_B$ is negative. The alignment of the two inputs is accomplished by shifting the mantissa of the smaller input the correct number of positions to the right. The number of positions to shift is determined by the difference of the two exponents (E_A and E_B). The sum can now be represented by:

$$A + B = (M_A \times 2^{E_A - E_B} + M_B) \times 2^{E_B}.$$

Where the two values in parentheses are of the same precedence. Figure 22 illustrates the general operations in a floating point adder.

The first block, the Zero Test block, is to determine if either of the two inputs are true zero. This is required only for representing the "hidden" bit of the input mantissas. If an operand is true zero then the "hidden" bit will be represented by zero. In this way the adder does not have any significant special handling logic for zero operands.

Before alignment can be completed, the greater exponent must be determined. This is done in the Exponent Compare block. This block provides the selection information to Mantissa Select block and selects the output sum's correct exponent. The simplest way to do this is to subtract one input exponent (E_B) from the other (E_A) and test the output for a positive number. If it is positive then A is has the largest exponent and E_A is the sum's exponent. If it is negative then B has the largest exponent and E_B is the sum's exponent. The outputs from this block are the sum's exponent that has not been adjusted yet and a selection bit to select which mantissa shall be aligned and which shall not. The Mantissa Select block merely selects which mantissa will be

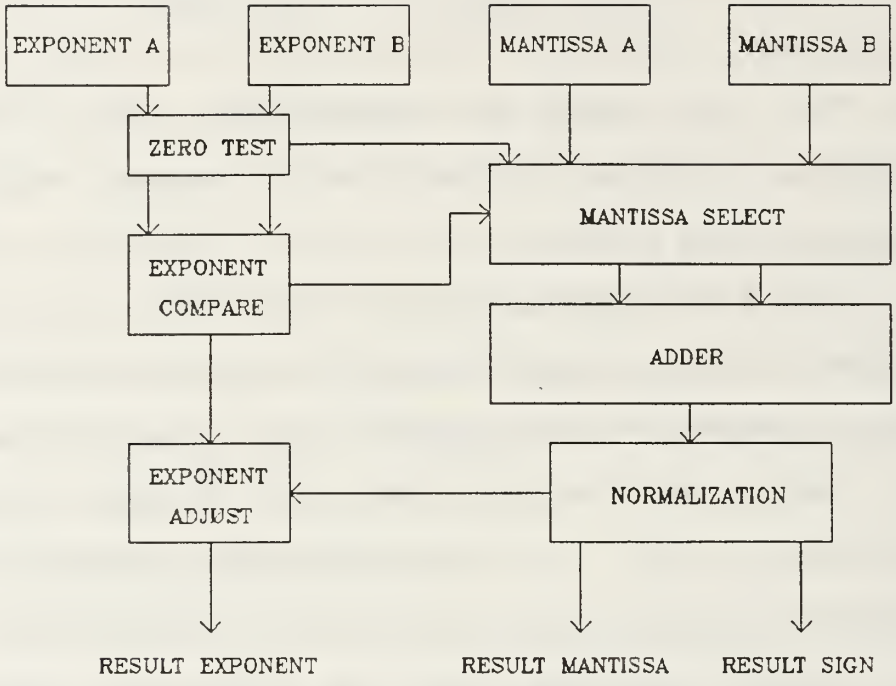


Figure 22. Block Diagram of a Floating Point Adder

shifted and completes the shift using the selection bit that is determined in the Exponent Compare block.

The Adder block completes the addition of the unshifted and shifted mantissas. The Normalization and the Exponent Adjust are similar to the Normalization and Exponent Adjust block of the multiplier allowing denormalized numbers.

E. SUMMARY

High speed arithmetic is useful for any application where fast computation is required. In signal processing applications, high speed arithmetic processing is a requirement. The fundamental building block of any spectrum analyzer is the FFT butterfly which is made from multipliers

and adders. The cyclic spectrum analyzer requires large FFTs to be computed which implies many multiplies and adds in the computation. To compute the cyclic plane in near real time, the multipliers and adders must be extremely fast. Chapter 3 describes the design of FFTs and cyclic spectrum analyzers in terms of number of FFT butterflies, multipliers and adders.

III. CYCLIC SPECTRUM ANALYZER

A. INTRODUCTION

The objective of this chapter is to consider ways to implement cyclic spectrum algorithms in near real time. The value used to characterize the closeness to real time is called the Real Time Factor (F_T):

$$F_T = \frac{COMPUTATION\ TIME}{COLLECT\ TIME}.$$

The number of hardware units (p_{hu}) needed to operate at a given factor of real time is:

$$p_{hu} = \frac{C_u}{F_T \Delta t} = \frac{C_u}{F_T N}.$$

Where $\Delta t = N$ is the total number of samples processed and C_u is the number of operations performed by the hardware unit. The complexity product ($p_{hu} * F_T$) is defined as a measure of the hardware complexity of the implemented algorithms (FFT butterflies and cyclic spectrum analyzers) to be discussed. [Ref. 1]

B. FFT DESIGN

1. Introduction

The fundamental building block for any spectrum analyzer is the FFT butterfly. There are two versions to

implement the FFT algorithm: Decimation in Time (DIT) and Decimation in Frequency (DIF). The following discussion addresses both versions of the radix-2 and radix-4 butterflies. N-point FFT designs using the radix-2 and radix-4 are discussed and compared.

2. Radix-2 FFT Butterfly

The radix-2 FFT butterfly is simply a method to compute the Discrete Fourier Transform (DFT) of a two point sequence. This DFT can be expressed as:

$$X(k) = \sum_{n=O}^{1} x(n)\, e^{-j2\pi nk/2},$$

letting $W_2 = e^{-j2\pi/2}$. $(W_2)^{nk}$ is called the weighting or twiddle factor. $X(k)$ is then equal to:

$$X(k) = x(0)\,(W_2)^{0k} + x(1)\,(W_2)^{1k}.$$

Substituting the value for W_2, $X(0)$ and $X(1)$ are:

$$X(0) = x(0) + x(1);$$

$$X(1) = x(0) - x(1).$$

The signal flow graph for this algorithm is shown in Figure 23 and is called the radix-2 FFT butterfly. This algorithm can be implemented with an inversion and 2 adds.

3. Radix-4 FFT Butterfly

The radix-4 FFT butterfly is a method to compute the DFT of a 4 point sequence. This DFT is expressed as:

$$X(k)=\sum_{n=0}^{3} x(n)\,(W_4)^{nk}=x(0)\,(W_4)^{0k}+x(1)\,(W_4)^{1k}+x(2)\,(W_4)^{2k}+x(3)\,(W_4)^{3k},$$

where $k = 0,1,2,3$ and $W_4 = e^{-j2\pi/4}$ and

$$(W_4)^0 = 1; \quad (W_4)^1 = -j; \quad (W_4)^2 = -1; \quad (W_4)^3 = j;$$

then $X(k)$ is:

$$X(k) = x(0) + x(1)\,(-j)^k + x(2)\,(-1)^k + x(3)\,(j)^k.$$

The signal flow graph for this algorithm is shown in Figure 24 and is called the radix-4 FFT butterfly.

4. N Point FFTs

a. Introduction

Obviously, sequences that are to be transformed are much longer that 2 or 4 points. They could be as long as a million or more points. The N point DFT is given as:

$$X(k) = \sum_{n=0}^{N-1} x(n)\,e^{-j\frac{2\pi}{N}nk}; \quad k = 0,1,2,\ldots,N-1.$$

This DFT can be realized with either the radix-2 or radix-4 FFT butterflies.

b. Radix-2 FFT

(1) Introduction. An $N = 2^\gamma$ point FFT can be constructed from radix-2 FFT butterflies. Let n and k be represented in binary form:

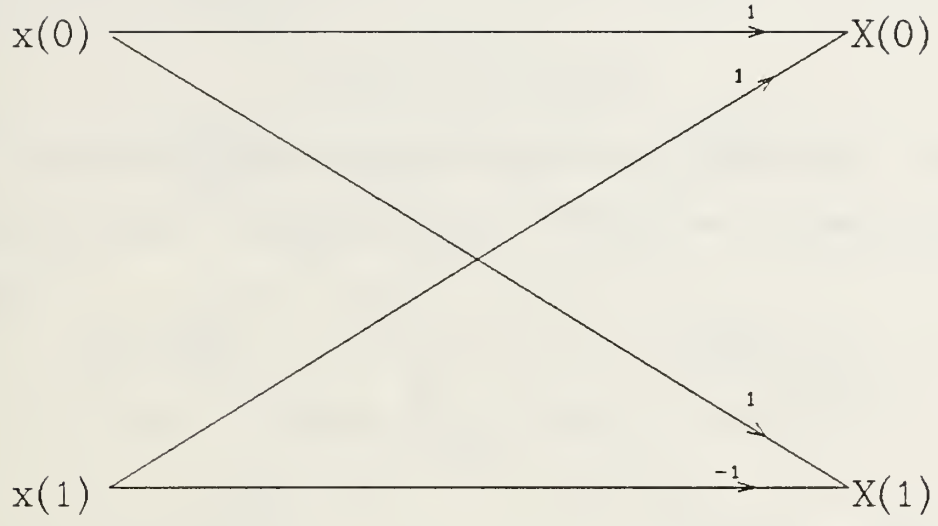


Figure 23. Radix-2 FFT Butterfly

$$n = 2^{\gamma-1}n_{\gamma-1} + 2^{\gamma-2}n_{\gamma-2} + \cdots + n_0;$$

$$k = 2^{\gamma-1}k_{\gamma-1} + 2^{\gamma-2}k_{\gamma-2} + \cdots + k_0;$$

then $X(k)$ is:

$$X(k_{\gamma-1}, \ldots, k_0) = \sum_{n_0=0}^{1} \sum_{n_1=0}^{1} \cdots \sum_{n_{\gamma-1}=0}^{1} x(n_{\gamma-1}, \ldots, n_0) W^p.$$

Where p is:

$$p = nk = (2^{\gamma-1}n_{\gamma-1} + \cdots + n_0)(2^{\gamma-1}k_{\gamma-1} + \cdots + k_0).$$

(2) DIT Algorithm. From the previous equation W^p can be rewritten as:

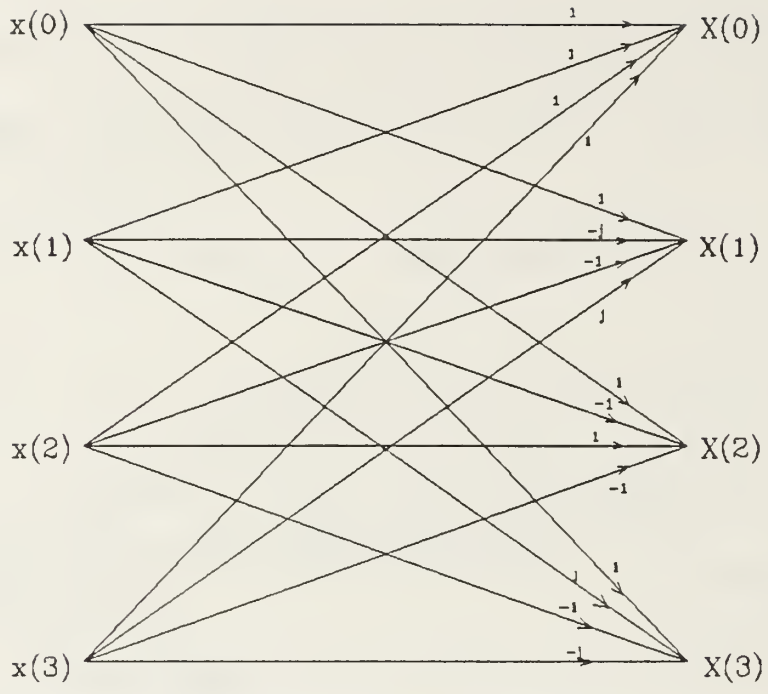


Figure 24. Radix-4 FFT Butterfly

$$W^{p} = [W^{(2^{\gamma-1}k_{\gamma-1} + \cdots + k_0)(2^{\gamma-1}n_{\gamma-1})}] \times \cdots \times [W^{(2^{\gamma-1}k_{\gamma-1} + \cdots + k_0)(n_0)}] .$$

The first term can be simplified into:

$$W^{(2^{\gamma-1}k_{\gamma-1} + \cdots + k_0)(2^{\gamma-1}n_{\gamma-1})} = W^{2^{\gamma-1}(k_0 n_{\gamma-1})},$$

and the second into:

$$W^{(2^{\gamma-1}k_{\gamma-1} + \cdots + k_0)(2^{\gamma-2}n_{\gamma-2})} = W^{(2k_1+k_0)(2^{\gamma-2}n_{\gamma-2})},$$

because:

$$W^{2^{\gamma}} = W^{N} = e^{-j\frac{2\pi}{N}N} = 1.$$

The rest of the terms are simplified in a similar manner. $X(k)$ can now be rewritten as:

$$X(k_{\gamma-1},\ldots,k_0) = \sum_{n_0}^{1}\cdots\sum_{n_{\gamma-1}}^{1} x(n_{\gamma-1},\ldots,n_0)\,[W^{2^{\gamma-1}(k_0 n_{\gamma-1})}]\,[W^{2^{\gamma-2}(2k_1+k_0)n_{\gamma-2}}]\times\cdots$$

$$\cdots\times[W^{2^{\gamma-1}k_{\gamma-1}+\cdots+k_0)n_0}.$$

Where the W's are the twiddle factors for each stage. The FFT can be written recursively as a function of the previous stages:

$$x_1(k_0,n_{\gamma-2},\ldots,n_0) = \sum_{n_{\gamma-1}=0}^{1} x_0(n_{\gamma-1},\ldots,n_0)\,W^{2^{\gamma-1}(k_0 n_{\gamma-1})};$$

$$x_2(k_0,k_1,n_{\gamma-3},\ldots,n_0) = \sum_{n_{\gamma-2}=0}^{1} x_1(k_0,n_{\gamma-2},\ldots,n_0)\,W^{2^{\gamma-1}(2k_1+k_0)n_{\gamma-2}};$$

and finally:

$$x_\gamma(k_0,\ldots,k_{\gamma-1}) = \sum_{n_0=0}^{1} x_{\gamma-1}(k_0,\ldots,k_{\gamma-2},n_0)\,W^{(2^{\gamma-1}k_{\gamma-1}+\cdots+k_0)n_0};$$

$$X(k) = X(k_{\gamma-1},\ldots,k_0) = x_\gamma(k_0,\ldots,k_{\gamma-1}).$$

Where $x_0(n)$ is the input sequence. The output of the final stage will always be in bit reversed order hence the bit reversal of the last equation. [Ref. 9:pp. 176-178]

(3) DIF Algorithm. W^p can be rewritten as:

$$W^p = [W^{(2^{\gamma-1}n_{\gamma-1}+\cdots+n_0)k_0}]\,[W^{(2^{\gamma-1}n_{\gamma-1}+\cdots+n_0)2k_1}]\cdots[W^{2^{\gamma-1}n_{\gamma-1}+\cdots+n_0)2^{\gamma-1}k_{\gamma-1}}].$$

Simplification similar to that done in the DIT algorithm leads to the following recursive equations that describe the FFT as a function of the previous stages:

$$x_1(k_0, n_{\gamma-2}, \ldots, n_0) = \sum_{n_{\gamma-1}}^{1} x_0(n_{\gamma-1}, \ldots, n_0) W^{(2^{\gamma-1}n_{\gamma-1} + \cdots + n_0)k_0};$$

$$x_2(k_0, k_1, n_{\gamma-3}, \ldots, n_0) = \sum_{n_{\gamma-2}=0}^{1} x_1(k_0, n_{\gamma-2}, \ldots, n_0) W^{(2^{\gamma-1}n_{\gamma-1} + \cdots + n_0)2k_1};$$

and finally:

$$x_\gamma(k_0, \ldots k_{\gamma-1}) = \sum_{n_0=0}^{1} x_{\gamma-1}(k_0, \ldots, k_{\gamma-2}, n_0) W^{2^{\gamma-1}n_0 k_{\gamma-1}};$$

$$X(k) = X(k_{\gamma-1}, \ldots k_0) = x_\gamma(k_0, \ldots k_{\gamma-1}).$$

[Ref. 9:pp. 177-180]

(4) Complexity of FFT using the Radix-2 Butterfly. The complexity for an N point, radix-2 FFT can be computed in number of butterflies. Each stage will require $N/2$ two point FFTs (butterflies). The total number of stages is $\log_2(N)$. The total number of radix-2 butterflies required to implement an N point FFT is:

$$C_B = \frac{N}{2}\log_2 N.$$

If the butterfly is implemented in a complex number system then the number of complex multiplies (C_{Bm}) and additions (C_{Ba}) are:

$$C_{Bm} = \frac{N}{2}\log_2 N;$$

$$C_{Ba} = N\log_2 N.$$

Then the number of real multiplies (C_{Brm}) and additions (C_{Bra}) is:

$$C_{Brm} = 2N\log_2 N;$$

$$C_{Bra} = 3N\log_2 N.$$

If each multiply and add were implemented in hardware then the FFT would be a rate-1 operator i.e., in each clock cycle, 2 complex x values and 1 W value would be input to the butterfly and the complex butterfly would produce 2 complex output values. However, it would be uneconomical and unwise to implement the FFT as a rate-1 operator with rate-1 butterflies. The radix-2 butterfly can be designed as a rate-1/2 operator which gives a complexity product for an N point FFT of:

$$p_{hu} \cdot F_T = \frac{2C_b}{N}.$$

Substituting C_B for C_b then:

$$p_{hu} \cdot F_T = \log_2 N.$$

This shows that to achieve real time ($F_T = 1$) then $p_{hu} = \log_2 N$ hardware units (complex 1/2 rate radix-2 FFT butterflies) are required. [Ref. 1] Reference 1 shows a structure of a radix-2 FFT constructed using rate-1/2 complex butterfly units.

c. Radix-4 FFT

(1) DIT Algorithm. An $N = 4^{\beta}$ FFT can be constructed from radix-4 FFT butterflies. Let n and k be represented in quaternary form:

$$n = 4^{\beta-1}n_{\beta-1} + 4^{\beta-2}n_{\beta-2} + \cdots + n_0;$$

$$k = 4^{\beta-1}k_{\beta-1} + 4^{\beta-2}k_{\beta-2} + \cdots + k_0;$$

then $X(k)$ is:

$$X(k_{\beta-1},\ldots,k_0) = \sum_{n_0=0}^{3}\sum_{n_1=0}^{3}\cdots\sum_{n_{\beta-1}}^{3} x(n_{\beta-1},\ldots,n_0)\,W^{p},$$

if $p = nk$, then W^{p} is:

$$W^{p} = [W^{(4^{\beta-1}k_{\beta-1} + \cdots + k_0)(4^{\beta-1}n_{\beta-1})}] \times \cdots \times [W^{(4^{\beta-1}k_{\beta-1} + \cdots + k_0)(n_0)}].$$

The first term can be simplified into:

$$W^{(4^{\beta-1}k_{\beta-1} + \cdots + k_0)(4^{\beta-1}n_{\beta-1})} = W^{4^{\beta-1}(k_0 n_{\beta-1})},$$

and the second into:

$$W^{(4^{\beta-1}k_{\beta-1} + \cdots + k_0)(4^{\beta-2}n_{\beta-2})} = W^{(4k_1+k_0)(4^{\beta-2}n_{\beta-2})},$$

because:

$$W^{4^{\beta}} = W^{N} = e^{-j\frac{2\pi}{N}N} = 1.$$

The rest of the terms are simplified in a similar manner. The FFT is written recursively as a function of the previous stages:

$$x_1(k_0, n_{\beta-2}, \ldots, n_0) = \sum_{n_{\beta-1}=0}^{3} x_0(n_{\beta-1}, \ldots, n_0)\, W^{4^{\beta-1} k_0 n_{\beta-1}};$$

$$x_2(k_0, k_1, n_{\beta-3}, \ldots, n_0) = \sum_{n_{\beta-2}=0}^{3} x_1(k_0, n_{\beta-2}, \ldots, n_0)\, W^{4^{\beta-2}(4k_1 + k_0) n_{\beta-2}};$$

and finally:

$$x_\beta(k_0, \ldots k_{\beta-1}) = \sum_{n_0=0}^{3} x_{\beta-1}(k_0, \ldots k_{\beta-2}, n_0)\, W^{(4^{\beta-1} k_{\beta-1} + \cdots + k_0) n_0};$$

$$X(k) = X(k_{\beta-1}, \ldots, k_0) = x_\beta(k_0, \ldots, k_{\beta-1}).$$

The twiddle factors of each stage can be simplified as follows:

$$W^{4^{\beta-1} k_0 n_{\beta-1}} = (W^{4^{\beta-1}})^{k_0 n_{\beta-1}} = (W^{\frac{N}{4}})^{k_0 n_{\beta-1}} = (e^{-j\frac{2\pi}{N}\frac{N}{4}})^{k_0 n_{\beta-1}} = (W_4)^{k_0 n_{\beta-1}};$$

$$W^{4^{\beta-2}(4k_1+k_0) n_{\beta-2}} = (W^{\frac{N}{16}})^{(4k_1+k_0) n_{\beta-2}} = (W_{16})^{(4k_1+k_0) n_{\beta-2}} = (W_4)^{k_1 n_{\beta-2}} (W_{16})^{k_0 n_{\beta-2}};$$

and finally:

$$W^{(4^{\beta-1} k_{\beta-1} + \cdots + k_0) n_0} = (W_4)^{k_{\beta-1} n_0} (W_{16})^{k_{\beta-2} n_0} \times \cdots \times (W_N)^{k_0 n_0}.$$

The W_4 terms are the twiddle factors internal to the radix-4 butterfly. All of the other W terms are external twiddle factors. These factors indicate that the first external twiddle factor of the butterfly will always be 1. This means that only three twiddle factor multipliers are required.

(2) DIF Algorithm. Using a similar simplification and rearrangement of the W exponents as in the radix-2 DIF algorithm, the recursive equations that describe the FFT as a function of the previous stages are:

$$x_1(k_0, n_{\beta-2}, \ldots, n_0) = \sum_{n_{\beta-1}=0}^{3} x_0(n_{\beta-1}, \ldots, n_0)\, W^{(4^{\beta-1}n_{\beta-1} + \cdots + n_0)k_0};$$

$$x_2(k_0, k_1, n_{\beta-3}, \ldots, n_0) = \sum_{n_{\beta-2}}^{3} x_1(k_0, n_{\beta-2}, \ldots, n_0)\, W^{(4^{\beta-1}n_{\beta-1} + \cdots + n_0)4k_1};$$

and finally:

$$x_\beta(k_), \ldots, k_{\beta-1}) = \sum_{n_0=0}^{3} x_{\beta-1}(k_0, \ldots, k_{\beta-2}, n_0)\, W^{4^{\beta-1}n_0k_{\beta-1}};$$

$$X(k) = X(k_{\beta-1}, \ldots, k_0) = x_\beta(k_0, \ldots, k_{\beta-1}).$$

The twiddle factors can be simplified for this algorithm also:

$$W^{(4^{\beta-1}n_{\beta-1} + \cdots + n_0)k_0} = (W_4)^{n_{\beta-1}k_0}(W_{16})^{n_{\beta-2}k_0} \times \cdots \times (W_N)^{n_0k_0};$$

$$W^{(4^{\beta-1}n_{\beta-1} + \cdots + n_0)4k_1} = (W_4)^{n_{\beta-1}k_1} \times \cdots \times (W_N)^{4k_1n_0} = (W_4)^{n_{\beta-2}k_1} \times \cdots \times (W_{N/4})^{k_1n_0}$$

and finally:

$$W^{4^{\beta-1}n_0k_{\beta-1}} = (W_4)^{n_0k_{\beta-1}}.$$

These equations also indicate only three external twiddle factor multiplies per butterfly. The difference is that the multiplies are after the butterfly instead of before the butterfly.

(3) Complexity of FFT using the Radix-4 Butterfly. The complexity for an N point, radix-4 FFT can be computed in number of butterflies. Each stage requires $N/4$ 4-point FFTs (butterflies), and the total number of stages is $\log_4(N)$. The total number of radix-4 butterflies required to implement an N point FFT is:

$$C_B = \frac{N}{4}\log_4 N.$$

If the butterfly is implemented in a complex number system then the number of complex multiplies (C_{Bm}) and additions (C_{Ba}) are:

$$C_{Bm} = \frac{3}{4}N\log_4 N;$$

$$C_{Ba} = 12N\log_2 N.$$

Then the number of real multiplies (C_{Brm}) and additions (C_{Bra}) is:

$$C_{Brm} = 3N\log_4 N;$$

$$C_{Bra} = \frac{51}{2}N\log_2 N.$$

If there is a multiplier and adder for each of these operations the FFT would be a rate-1 operator. As in the case of the FFT implemented with radix-2 butterflies, implementing the FFT in this manner would be uneconomical. The radix-4

butterfly is best implemented as a rate-1/4 operator which gives a complexity product for an N point FFT of:

$$p_{hu} \cdot F_T = \frac{4C_b}{N}.$$

The rate-1/4 operator is shown schematically in Figure 25. It receives 4 complex x-inputs sequentially in 4 clock cycles and simultaneously inputs 4 multiplying complex twiddle factors and produces the 4 next level components sequentially after a pipeline delay of d clock cycles. Substituting C_B for C_b then:

$$p_{hu} \cdot F_T = \log_4 N.$$

This shows that to achieve real time (F_T = 1) then $p_{hu} = \log_4 N$ hardware units (complex 1/4 rate radix-4 FFT butterflies) are required. [Ref. 1]

d. Comparison

The complexity of an FFT using the radix-2 and radix-4 complex FFT butterflies is given in paras. B.4.b.4 and B.4.c.3, respectively. Although the complexity of the radix-4 butterfly is much greater than the radix-2 butterfly, an FFT built with radix-4 butterflies has many less butterfly units than the FFT built with radix-2 butterflies ($(N/4)\log_4 N$ as opposed to $(N/2)\log_2 N$). If the complex radix-4 butterfly can be built on 1 chip then a large FFT should be built with these units.

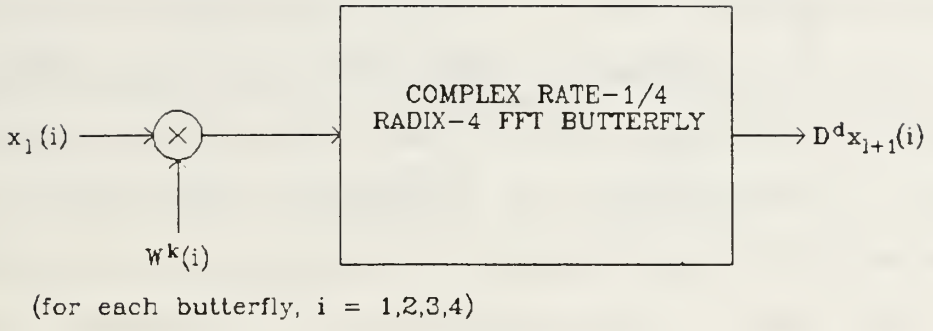


Figure 25. Rate-1/4 Complex Radix-4 FFT Butterfly

C. CYCLIC SPECTRUM ANALYZER DESIGN

1. Input Specifications

The input to the analyzer is a sequence of floating point values that are obtained by sampling a real wideband signal at approximately 50 MHz and then applying the Hilbert transform to the digital signal. This implies a bandwidth of at most 50 MHz using the complex envelope. Since the signals of interest (SOI) for this analyzer are primarily wideband signals, the frequency resolution is not required to be extremely small.

2. Digital Implementation

a. Introduction

There are three algorithms to compute the cyclic spectrum stated in Chapter 1: 1) Frequency Smoothing Method (FSM), 2) Strip Spectral Correlation Analyzer (SSCA), 3) Frequency Accumulation Method (FAM). All of these algorithms require a large number of arithmetic computations to be implemented and a large amount of hardware to execute them in near real time. The objective is to consider ways to implement these algorithms in real time. [Ref. 1]

b. Frequency Smoothing Method (FSM)

The FSM algorithm consists of two parts: an N point spectral correlator and an M point summation unit. M is the time-bandwidth product ($\Delta t * \Delta f$). Figure 26 illustrates the architecture for the FSM algorithm. The frequency values f and α are denoted by:

$$f = \frac{k + l}{2N};$$

$$\alpha = \frac{k - l}{N}.$$

Where N is the length of the input sequence, and k and l are sequence indices. The complexity of this algorithm, described in number of FFT butterflies is:

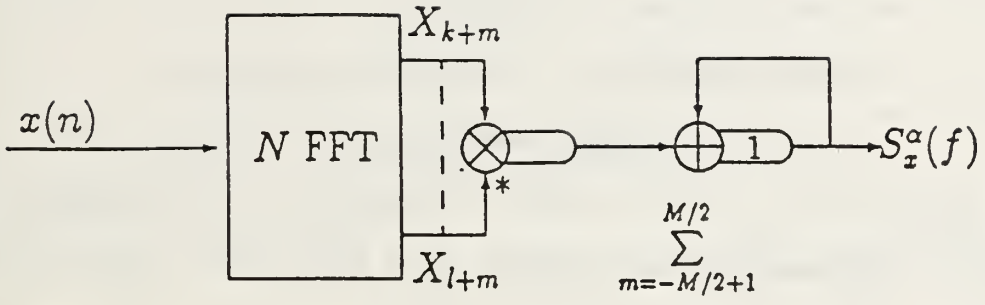


Figure 26. Frequency Smoothing Method Architecture

$$C_b = \frac{\Delta t}{2} \log_2 \Delta t$$

complex radix-2 butterflies and the complexity product is:

$$p_{b2} \cdot F_T = \log_2 \Delta t$$

for rate-1/2 complex radix-2 FFT butterflies. Implementing the algorithm's FFTs with radix-4 butterflies, the complexity product is:

$$p_{b4} \cdot F_T = \log_4 \Delta t.$$

The correlator portion has a requirement for separate multipliers enumerated by the complexity product $p_{m1} * F_T$ given by:

$$p_{rm1} \cdot F_T = \frac{\Delta t \Delta f}{\Delta f}$$

rate-1 real multipliers. [Ref. 1]

c. Strip Spectral Correlation Analyzer

The SSCA algorithm consists of four parts: 1) an N' point FFT, 2) a down conversion multiplier, 3) a correlation multiplier, and 4) an N point FFT. Where $LP = \Delta t$, $L = N'/4 = N/4M$, $\Delta f = 1/N' = M/N$, $\Delta\alpha = 1/\Delta t = 1/N$, and $\Delta f \Delta t = N/N' = M$. Figure 27 [Ref. 1] illustrates the architecture for the SSCA algorithm. [Ref. 1]

The complexity of this algorithm, described in number of FFT butterflies is:

$$C_{b2} = \frac{\Delta t \Delta f}{4\Delta f^2} \log_2 \frac{\Delta t \Delta f}{\Delta f} + \frac{2\Delta t \Delta f}{\Delta f} \log_2 \frac{1}{\Delta f}.$$

Then the complexity product is:

$$p_{b2} \cdot F_T = \frac{1}{2\Delta f} \log_2 \frac{\Delta t \Delta f}{\Delta f} + 4 \log_2 \frac{1}{\Delta f}$$

for the rate-1/2 complex butterflies. Using radix-4 butterflies, the complexity product is derived from the complexity of the FFTs:

$$C_{b4} = N \log_4 \frac{N}{M} + \frac{N^2}{8M} \log_4 N = \frac{\Delta t \Delta f}{\Delta f} \log_4 \frac{1}{\Delta f} + \frac{\Delta t \Delta f}{8\Delta f^2} \log_4 \frac{\Delta t \Delta f}{\Delta f}.$$

Then the complexity product is:

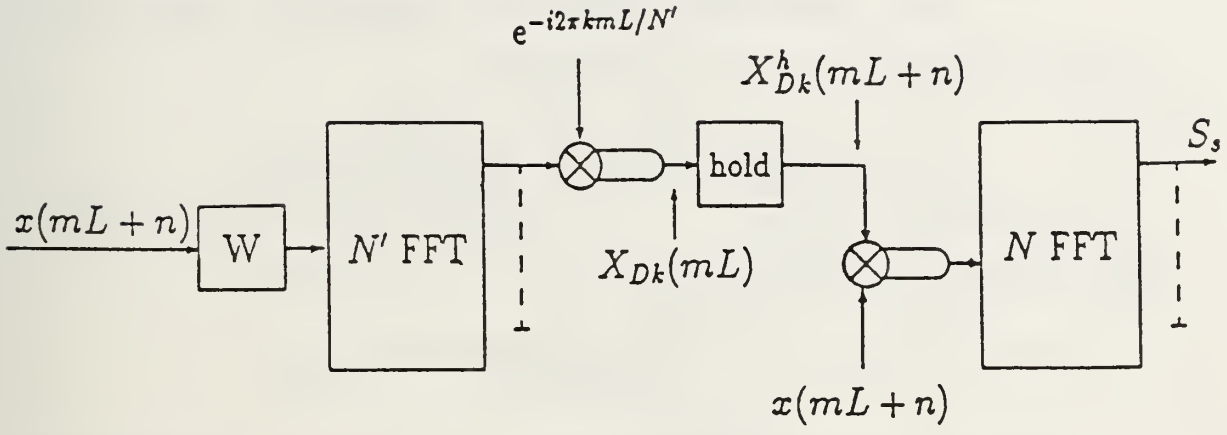


Figure 27. Strip Spectral Correlation Analyzer Architecture

$$p_{b4} \cdot F_T = \frac{4C_{b4}}{N} = \frac{4C_{b4}}{\Delta t} = 4\log_4\frac{1}{\Delta f} + \frac{1}{2\Delta f}\log_4\frac{\Delta t\Delta f}{\Delta f}$$

for rate-1/4 complex radix-4 butterflies. The remaining number of multiplies is:

$$p_{rm} \cdot F_T = \frac{2}{\Delta f} + 12$$

for the rate-1 multipliers. [Ref. 1]

d. FFT Accumulation Method (FAM)

The FAM algorithm consists of four parts: 1) an N' point FFT, 2) a down conversion, 3) a correlation multiplier, and a P point FFT. Where $N = \Delta t = LP$, $1/N' = M/N = \Delta f$, and L

= $N'/4$. Figure 28 illustrates the architecture of the FAM algorithm. [Ref. 1]

The complexity of this algorithm, described in number of FFT butterflies is:

$$C_{b2} = \frac{2\Delta t\Delta f}{\Delta f}\log_2\frac{1}{\Delta f} + \frac{\Delta t\Delta f}{2\Delta f^2}\log_2 4\Delta t\Delta f.$$

The complexity product is:

$$p_{b2} \cdot F_T = \frac{1}{\Delta f}\log_2 4\Delta t\Delta f + 4\log_2\frac{1}{\Delta f}$$

for rate-1/2 complex butterflies. Using radix-4 butterflies, the complexity product is derived from the complexity of the FFTs:

$$C_{b4} = N\log_4\frac{N}{M} + \frac{N^2}{4M}\log_4 4M = \frac{\Delta t\Delta f}{\Delta f}\log_4\frac{1}{\Delta f} + \frac{\Delta t\Delta f}{4\Delta f^2}\log_4 4\Delta t\Delta f.$$

Then the complexity product is:

$$p_{b4} \cdot F_T = \frac{4C_{b4}}{N} = 4\log_4\frac{1}{\Delta f} + \frac{1}{\Delta f}\log_4 4\Delta t\Delta f$$

for rate-1/4 complex radix-4 butterflies. The remaining number of multiplies is:

$$p_{rm} \cdot F_T = \frac{4N}{M} + 20 = \frac{4}{\Delta f} + 20$$

for the rate-1 real multipliers associated with the down conversion, the correlator, and the windowing function. [Ref. 1]

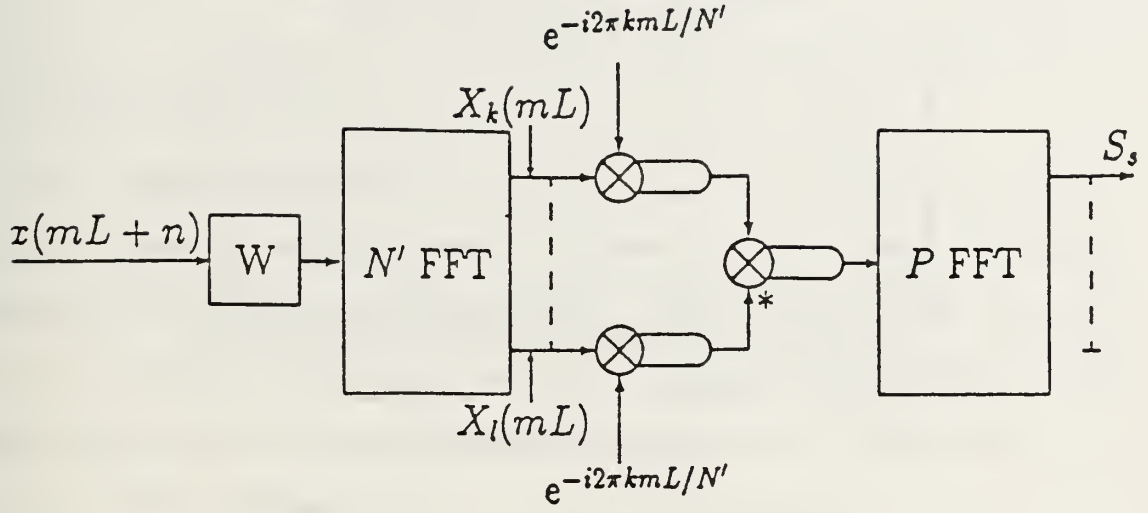


Figure 28. FFT Accumulation Method Architecture

D. CONCLUSIONS

Figure 29 [Ref. 1] is a log-log plot of the complexity product versus time-bandwidth product ($\Delta t \Delta f$) for a given Δf = 1/8. Although the FSM algorithm is the simplest conceptually, the algorithm is obviously much more complex than the two other algorithms. The SSCA algorithm has the smallest complexity for large N. Figure 30 is log-log plot of the complexity product versus $\Delta t \Delta f$ with a given Δf = 1/8 for the FAM algorithm using the radix-2 and the radix-4 complex butterflies. Chapter 4 discusses the actual VLSI design of a rate-1/4 radix-4 complex FFT butterfly and the multiplier and adder that is used to construct it.

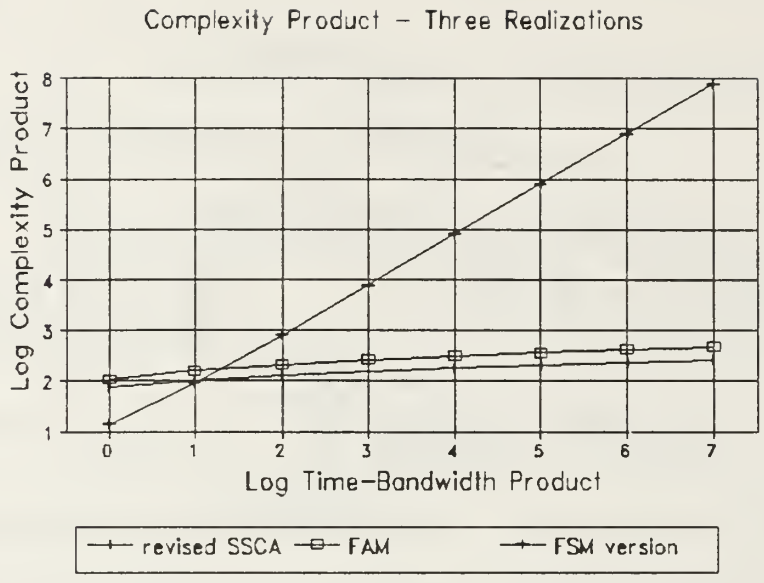


Figure 29. Log Complexity Product vs. Log $\Delta t \Delta f$

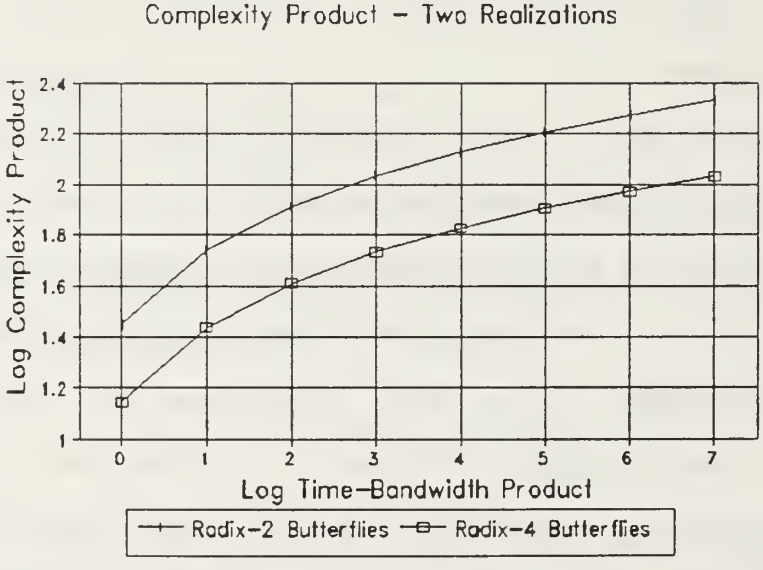


Figure 30. Log Complexity Product vs. Log $\Delta t \Delta f$ using Radix-2 and Radix-4 Butterflies

IV. DIGITAL DESIGNS

A. INTRODUCTION

1. Specifications

Specifications for the multiplier, adder, and FFT butterfly are given in Appendix B. The fabline MHS CN10C is a 1.0 micron feature size technology. To achieve the operating frequency required all designs are pipelined.

2. Pipelining

Pipelining is based on separating a logic circuit (multiplier, adder, or FFT) into smaller and faster subcircuits. These subcircuits or stages are separated by storage registers. The storage registers synchronize and save the output of one stage and provide that output as input into the next stage. Figure 31 illustrates a pipelined logic circuit. Although the delay from a given input to the correct output is longer and takes multiple clock cycles to complete one operation, this method provides a way to implement the logic circuit with a high frequency clock. New inputs must be provided into the pipelined circuit every clock cycle to keep the pipeline full. Pipelining is ideal for signal processing applications because the large amount of data to process will always keep the pipeline full.

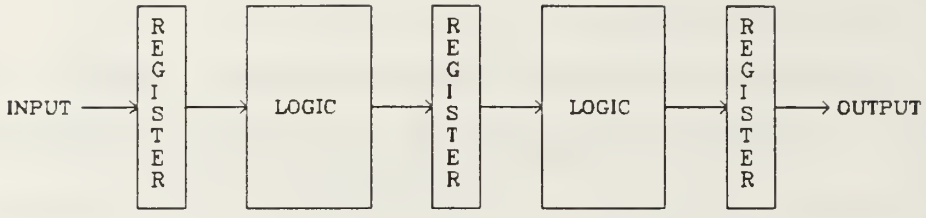


Figure 31. Pipelined Process

B. FLOATING POINT MULTIPLIER

1. Introduction

The floating point multiplier design had required only 6 stages to implement a 45 MHz clock frequency. By far the most limiting portion (the slowest stage) was the parallel multiplier array. Stage 1 is comprised of the Genesil library parallel multiplier, the conversion of the input exponents from excess code to two's complement, and the XOR of the input sign bits. Stage 2 sums the partial products from the parallel multiplier cell and add the two's complement exponents together. Stage 3 performs the normalization. Stage 4 performs the adding for the rounding function. Stage 5 performs the postnormalization required due to the addition

from rounding and the exponent adjusting due to normalization and postnormalization. Stage 6 provides the setting or clearing of all bits in the cases of exponent overflow or underflow. The multiplier will be described in the following paragraphs as illustrated in Figure 21 without regard to pipelining stage boundaries.

2. Multiplier Block

This block is the hardware that computes the product of the input mantissas. Since the mantissas each have a word length of 14 bits including the hidden bit, the product will be 28 bits wide. In this case, the Genesil library parallel multiplier provides the 14 least significant bits of the product but only provides two partial products for the next 13 more significant bits. These partial products must be summed together to get the 14 most significant bits of the product. The conditional sum adder provides the required speed with an acceptable amount of hardware used. The Genesil library multiplier also provides pregenerated "sticky" bits required for rounding. Sticky[1] is the OR of the least significant 12 product bits and sticky[0] is the OR of the least significant 13 product bits. The output of the Multiply block is the 16 most significant product bits and the "sticky" bits. The 12 least significant bits are adequately represented by the 2 most significant of these and by the "sticky" bits. Figure 32

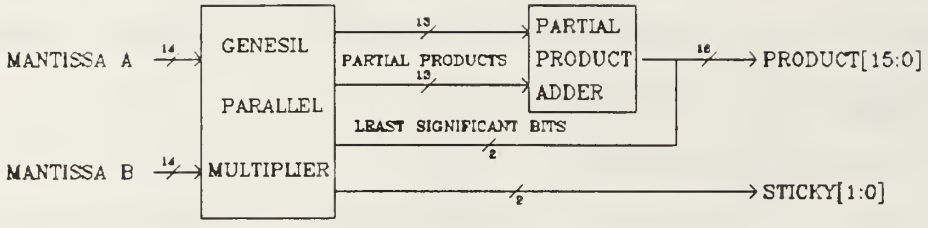


Figure 32. Multiply Block

is a block diagram illustrating the functions completed in the Multiplier block.

3. Exponent Add Block

The function of this block is to add the two input exponents. This would be rather difficult to do leaving the exponents in excess code, so the they are converted to two's complement simply by inverting the most significant bit of each of the exponents. Before the conversion is done, each exponent is tested for all zeros which indicates that the floating point number associated with the exponent is true zero. A flag is generated for this condition so that after all the operations are complete in the floating point multiply

the output can be set to zero. The exponents are then added as two's complement integers using a 6-bit carry ripple adder with additional logic to detect overflow and underflow. Overflow and underflow are indicated at the output of the floating point multiplier and they are also used to set or clear the output in the cleanup stage. The product exponent is then converted to excess code before being output from this block. Figure 33 is a block diagram illustrating the function of the Exponent Add block.

4. Normalization Block

a. Introduction

As stated in Chapter 2 the Normalization block can be broken down into 3 sub-blocks. Figure 34 is the block diagram describing the functions executed by the Normalization block. The Normalizer sub-block performs the initial normalizing of the mantissa product. The Rounder sub-block performs the rounding of the mantissa product to the correct number of significant bits. The Postnormalization sub-block performs the final normalization due to possible carry-out during the rounding process.

b. Normalizer Sub-Block

The Normalizer sub-block uses the 16-bit product output from the Multiplier to perform the initial normalization of the mantissa product. Since the number system only allows normalized numbers the input mantissas will

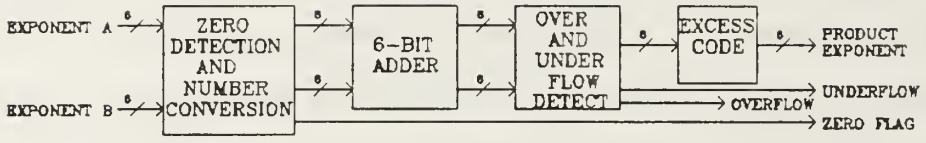


Figure 33. Exponent Add Block

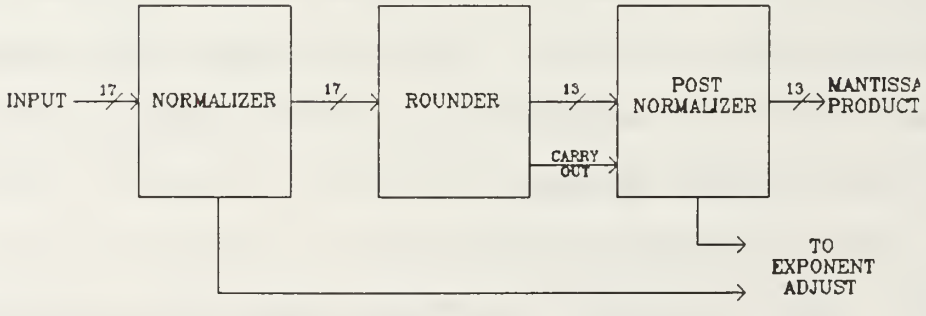


Figure 34. Normalization Block

always be between 1 and 2 and the product mantissa will always be between 1 and 4. This implies that normalization is only required when the product mantissa is greater than or equal to 2 or 10_2. This occurs when the most significant bit of the product mantissa is a 1. If this is the case, the product mantissa is shifted 1 bit to the right and a 1 is sent to the Exponent Adjust block to be added to the product exponent. If the most significant bit of the mantissa is a 0, then no normalization is done. In either case, the most significant bit is dropped since it is the hidden bit of the product mantissa. All of these operations are simply completed with a 2-input multiplexer. The output of the Normalizer sub-block is the 14-bit product mantissa and the add bit for the Exponent Adjust block.

c. Rounder Sub-Block

The Rounder performs the unbiased rounding of the product mantissa. The sticky bits and the two least significant bits of the product mantissa from the Normalizer sub-block are used through logic modules to determine when to add 1 to the 13 most significant bits of the product mantissa. The 13-bit product mantissa and the carry-out possibly generated is output to the Postnormalizer sub-block to complete the required normalization.

d. Postnormalizer Sub-Block

The Postnormalizer sub-block performs the same function as the Normalizer sub-block with the carry-out from the Rounder sub-block as the decision maker for normalizing. The output of the Postnormalizer sub-block is the final 13-bit product mantissa and another add bit for the Exponent Adjust block.

5. Exponent Adjust Block

The Exponent Adjust block is merely a 6-bit carry-ripple adder to add the excess code product exponent to the add bits generated in the Normalization Block. Since these add bits are only 1-bit wide they can both be added using one input and the carry-in of the least significant bit of the adder. Overflow during this add is detected by a carry-out from the adder. Underflow is not possible because the function is always an add, not a subtraction. The output of the Exponent Adjust block is the final product exponent and an overflow bit.

6. Clean-up

The Clean-up function is just clearing or setting every bit except the sign bit of the product for certain special cases. If there was exponent overflow either in the Exponent Add or Exponent Adjust blocks then all of the bits are set and output overflow bit is set. If there is exponent underflow, then all of the bits are cleared and the underflow

bit is set. In the case when at least one of the floating point inputs is zero, then all of the bits are cleared also.

C. FLOATING POINT ADDER

1. Introduction

The Floating Point Adder is a much more complex design than the multiplier. The only similarities between the multiplier and the adder are the Postnormalizer, the general structure of the Normalization blocks, and the final clean up. The adder design required 14 pipeline stages to implement it with a operating frequency over 45 MHz. No single function or section of the adder incurred the largest delay. The adder will be described in the following paragraphs as illustrated in Figure 22 without regard to pipelining stage boundaries.

2. Zero Test Block

The Zero Test block tests each input exponent for true zero. If an exponent is true zero then the Zero Test block generates a 0 for the "hidden" bit of its mantissa to be passed to the Mantissa Select block. The output of the Zero Test block is the input exponents and the 2 "hidden" bits.

3. Exponent Compare Block

The Exponent Compare block determines which floating point input number has the smaller exponent. After the exponents are converted to two's complement, this is accomplished by subtracting one from the other and vice versa

and checking the sign of one of the differences. Let A be one input and B the other and their associated exponents E_A and E_B. If $E_A - E_B < 0$ then E_A is the smaller exponent, so the Mantissa Select block is signaled to select the mantissa of A for right shifting and the difference is the number of bits to shift. E_B is selected as the sum exponent of the floating point number. The outputs of the Exponent Compare block are the sum exponent, the mantissa select bit, and the number of shifts. Figure 35 is a block diagram of the functions completed in the Exponent Compare block.

4. Mantissa Select Block

a. Introduction

The Mantissa Select block performs the overall function of shifting the mantissa of the floating point number with the smallest exponent to the right the correct number of bits to align it with the other mantissa. Figure 36 is a block diagram of the Mantissa Select block.

b. Ones' Conversion Sub-Block

Since the floating point numbers are in signed magnitude, the easiest and least hardware intensive number system to accomplish the sum is ones' complement. The conversion requires only to invert all the bits of the mantissa if the number is negative. This sub-block also concatenates the "hidden" bit as determined in the Zero Test

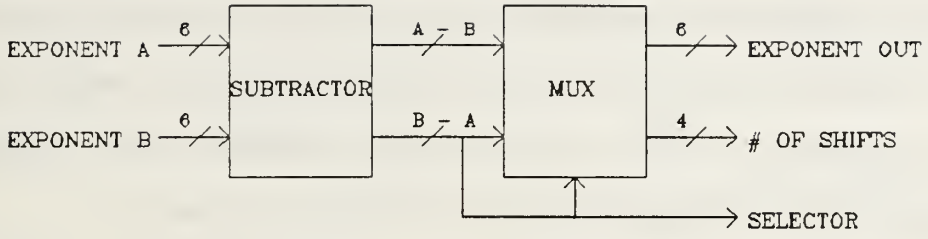


Figure 35. Exponent Compare Block

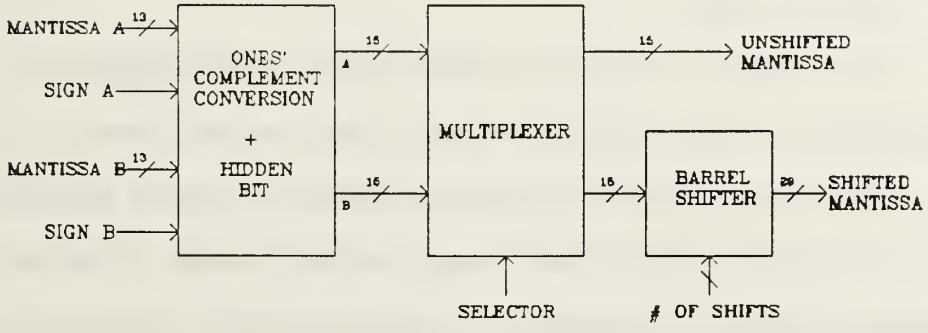


Figure 36. Mantissa Select Block

block. The outputs of the Ones' Conversion sub-block are 2 15-bit mantissas.

c. Selector Sub-Block

The Selector sub-block determines, using the mantissa select bit generated in the Exponent Compare block, which mantissa will be shifted and which will not. This is accomplished with two 2-input multiplexers. The outputs of the Selector sub-block are the two mantissas.

d. Align Sub-Block

The Align sub-block shifts the mantissa selected for alignment done in the Selector sub-block. The number of bit positions shifted to the right was determined in the Exponent Compare block. This shifting function is done with a barrel shifter. A barrel shifter uses logic not sequential circuitry to complete a shift. The output of the Align sub-block is the 29-bit shifted mantissa.

5. Adder Block

The Adder block performs the actual addition of the two mantissas. The integer adder used is the Conditional Sum adder. It provides the requisite speed without using a large amount of chip area. The most significant 15 bits of the shifted mantissa is added to the unshifted mantissa. Both mantissas are sign extended one bit to prevent carry-out due to overflow or underflow. Thus, the only reason for a carry-out would be to generate an "end-around" carry. The addition

of the "end-around" carry is done with 16-bit half-adder since the only inputs are the sum and the 1-bit "end-around" carry. The output of the adder block is the sum of the mantissas concatenated with the 15 least significant bits of the shifted mantissa. Figure 37 is a block diagram of the Adder block.

6. Normalization Block

a. Introduction

The Normalization block of the floating point adder is similar in structure to the one in the floating point multiplier. It can also be broken into the Normalizer, Rounder, and Postnormalizer sub-blocks. It differs only in the Normalizer sub-block since the mantissa sum could be any number between 0 and 4. This means the leading nonzero bit could be in any bit position. The Normalizer must be able to detect and shift accordingly. Then Postnormalization sub-block is exactly the same as the one in the floating point multiplier.

b. Normalizer Sub-Block

Before the mantissa sum can be normalized it must be converted from ones' complement to signed magnitude. Then the Normalizer sub-block uses a programmable priority encoder to sense the leading nonzero bit. This priority encoder can generate any bit pattern for a given bit position of the leading nonzero bit position. The pattern in this case is the number of bit positions to shift mantissa. But the exponent

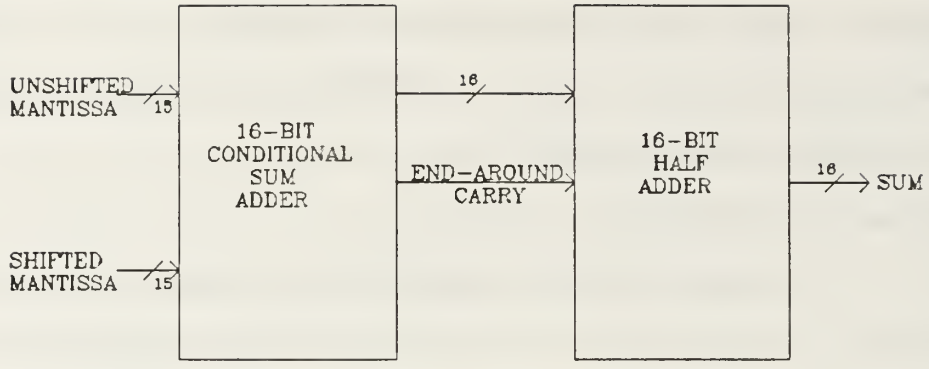


Figure 37. Adder Block

must be adjusted also, so another encoder is used to generate the two's complement number to be added to the exponent in the Exponent Adjust block. The mantissa will be shifted with a barrel shifter. The outputs of the Normalizer sub-block are the two's complement number to be added to the exponent, the 29-bit normalized mantissa sum, and the sign bit.

c. Rounder Sub-Block

The Rounder sub-block must determine from the 17 least significant bits with logic whether to round the mantissa sum up or to truncate. The outputs of the Rounder sub-block are rounded 13 mantissa sum bits and the carry-out possibly generated by the addition due to round up.

7. Exponent Adjust Block

The Exponent Adjust block provides the adder to add the two's complement number and the postnormalization exponent adjustment bit generated in the Normalization block to the exponent selected for output. Since the exponent is only 6 bits wide, the carry-ripple adder was used. The output of this block is the exponent of the final sum.

D. RADIX-4 FFT BUTTERFLY

1. Introduction

The design is a rate-1/4 radix-4 complex floating point FFT butterfly. Rate-1/4 implies that one data point is input and one FFT point is output every clock cycle. The butterfly can be separated into four parts: an external twiddle factor multiplier; a shift register and latch; an internal twiddle factor multiplier; and a 4-input adder. The external twiddle factor at the input implies the DIT algorithm. Figure 38 is the block diagram of the rate-1/4 FFT butterfly.

2. External Twiddle Factor Multiplier

The twiddle factor multiplier is just a complex multiplier to facilitate computation of large FFTs with the radix-4 butterfly. The complex multiplier will require 4 floating point multipliers and 2 floating point adders to implement. Figure 39 is the block diagram of the external twiddle factor multiplier.

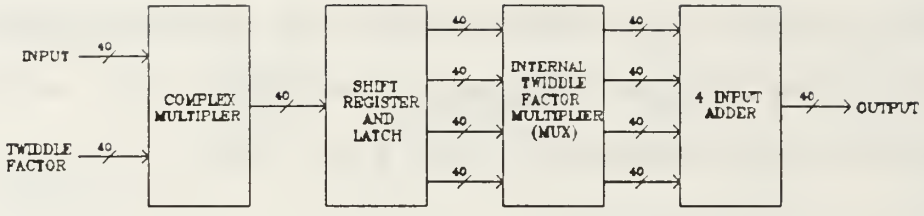


Figure 38. Rate-1/4 Radix-4 FFT Butterfly

3. Shift Register and Latch

To compute a 4-point FFT at 1/4 rate, 4 data points must be clocked in and held for four more clock cycles. Figure 40 is the block diagram of the shift register and latch, and the internal twiddle factor multiplier (multiplexers). The shift register portion is just 3 standard D-registers with their outputs connected to the inputs of the next register and to the inputs of a latch. A modulo-4 counter allows the 4 data points to be clocked in and when the counter goes from 11_b to 00_b it generates a carry-out which is ANDed with the PHASE_X clock and the output strobes the latch. The latch holds the four data points for 4 clock

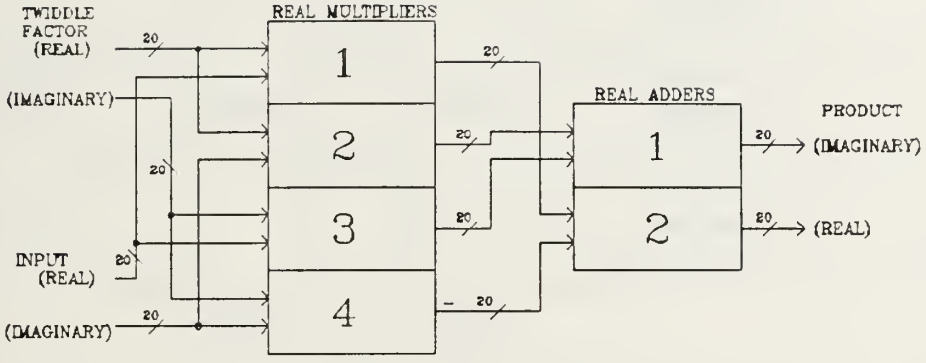


Figure 39. External Twiddle Factor Multiplier

cycles at such time the latch is strobed again to hold 4 more data points.

4. Internal Twiddle Factor Multiplier

The equation for the 4-input DFT computation was given in Chapter 3:

$$X(k) = x(0) + x(1)(-j)^k + x(2)(-1)^k + x(3)(j)^k \qquad k = 0,1,2,3.$$

To generate the correct summands for a given k, multiplexers and logic will be used. The first summand ($x(0)$) is constant with respect to k, so it is left unchanged. The second and fourth summands ($x(1)$ and $x(3)$) require 4-input multiplexers to complete the product. The third summand ($x(2)$) is just

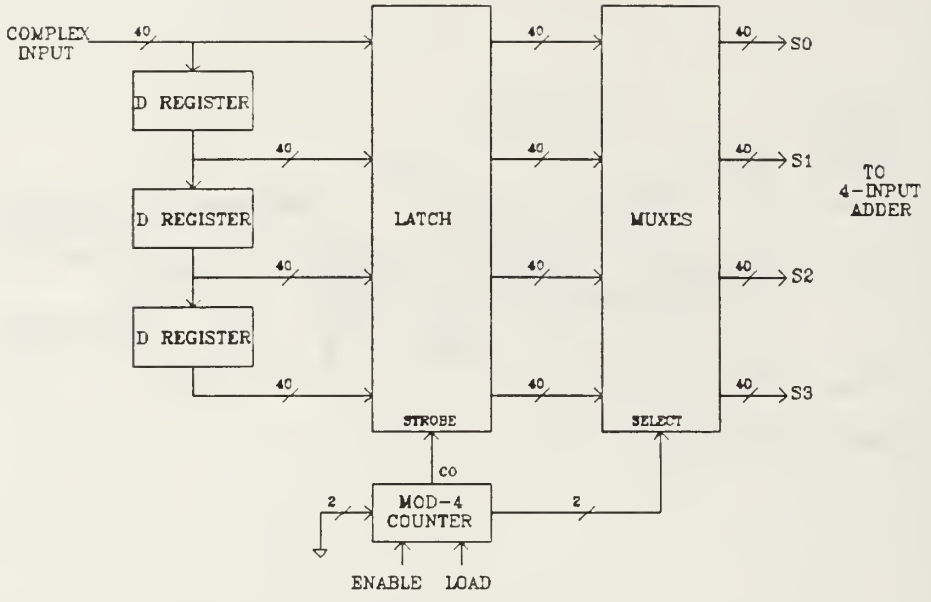


Figure 40. Shift Registers and Latch + Multiplexers

inverted for odd k. The different k's are generated by the counter in the shift register and latch portion.

5. 4-Input Complex Adder

This adder is just 3 2-operand complex floating point adders arranged as in Figure 41. Each 2-operand complex adder requires 2 2-operand real floating point adders which gives a total of 6 2-operand real floating point adders to implement the 4-input complex floating point adder.

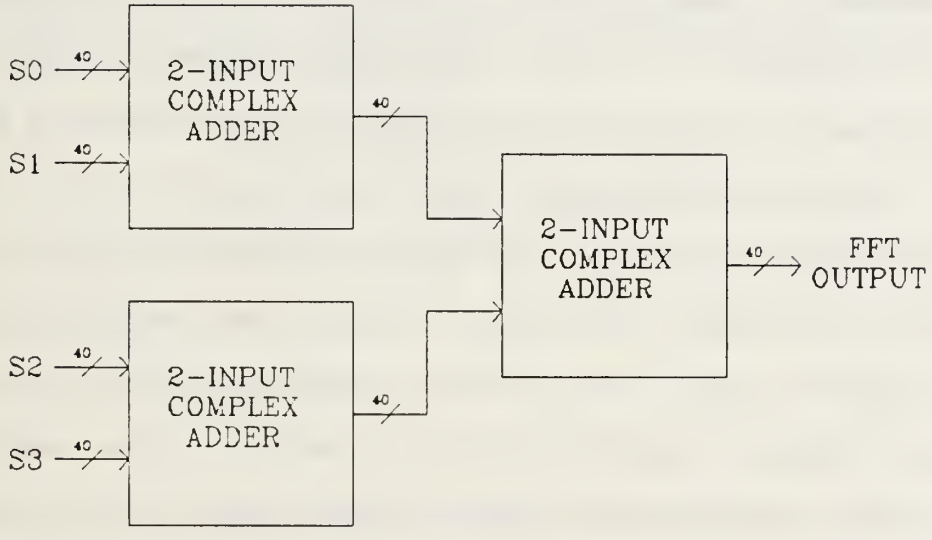


Figure 41. 4-Input Complex Adder

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The rate-1/4 radix-4 complex floating point FFT butterfly was successfully designed and simulated in VLSI at a clock frequency of approximately 45 MHz. The design is large and will be expensive to fabricate. It utilizes 4 floating point multipliers, 8 floating point adders, 2 4-input multiplexers, 3 data registers, 1 latch, and some assorted logic. The silicon area of the IC including pads is approximately 200,000 $mils^2$. Appendix B describes the IC in detail.

Logic-Compiler made this design feasible because of its ability to optimize the design for area and performance. If Logic-Compiler were not available and the author had to rely on the Genesil Standard layout compiler, the adder and multiplier would each be about 40,000 $mils^2$ in area and have an operating speed of less than 40 MHz. A complex multiplier implemented on one IC chip would not have been possible, let alone a radix-4 FFT butterfly. Logic-Compiler has made the Genesil Standard Compiler obsolete except for IC chip floorplanning which requires user floorplanning for pad placement.

B. RECOMMENDATIONS

The author makes the following recommendations:

- Investigate further the commercially available IC chips for FFT computations and control path design.
- Purchase 1.0 micron radiation hardened fabrication line library for Genesil.
- Investigate fabrication costs for the FFT butterfly IC chip.
- Begin design of chip sets for large FFTs and ultimately the cyclic spectrum analyzer.
- Design a bonafide 4 input adder to replace the one built from 3 2-input adders which will reduce silicon area of the FFT butterfly design.

APPENDIX A. AUTOLOGIC

A. GENESIL SILICON COMPILER

1. Introduction

The Genesil Designer is an integrated set of automated ASIC design tools that contain the IC design expertise necessary to transform a functional specification into a data base from which an IC can be produced. Genesil provides 1) High-level design entry allowing system designers to create dense physical designs for integrated circuits; 2) Rapid feedback on key performance metrics during exploratory and detailed design stages; 3) Verification tools for simulation, timing analysis, and layout; 4) Compiler libraries that can be expanded with user developed compilers; 5) The ability to import layouts designed with other CAD tools; 6) Multiple fabrication options for process-independence of designs. Figure 42 [Ref. 10:p. 1-1] illustrates the Genesil environment. Input to Genesil is done with high-level functional descriptions, using a combination of forms-based entry and schematic capture. Output consists of layout files that are sent to an IC manufacturer for fabrication.

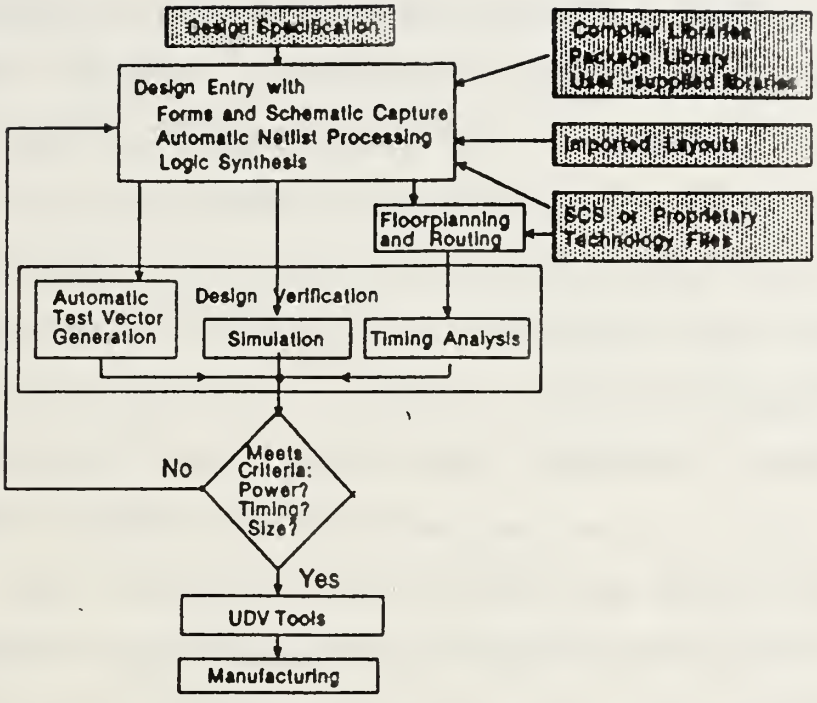


Figure 42. Genesil Environment

2. Design Process

a. Introduction

There are three phases in the Genesil design process: 1) Design Entry; 2) Design Verification; 3) Design Manufacture. Prior to beginning the design process, the designer must determine the required logic functionality, the physical requirements for timing, size, and power consumption, and the testability requirements.

b. Design Entry

(1) Introduction. Design entry in Genesil is called forms-based entry. Basically, the design parameters are input via menus and forms that Genesil provides. The

design process is initialized by selecting the type of module: Parallel Datapath, Block, Random Logic, General. Random Logic modules include functions that range from logic gates to multiplexers and full adder cells. Blocks include RAM, ROM, PLAs, pads, and parallel multiplier cells. Parallel Datapath modules are optimized for parallel data and control operations, including: arithmetic and logic functions; bus-structured interface operations; and parallel control operations. General modules contain a number of sub-modules that are of any type module described above. [Ref. 11:p. 1-1] There are four basic tasks to be accomplished in design entry; header definition, specification definition, netlisting, and floorplanning.

(2) Header Definition. The header form provides the Genesil user input to specify fabrication line, function type, IC package type, and compiler type. The fabrication line can be selected from the list provided in the header form. Header forms for Block module provides the different functions that can be selected. If the design is for a complete chip, the header form provides different packages in the Genesil library for selection. Compiler type can either be Standard Genesil or Logic-Compiler.

(3) Specification Definition. Definition of the logical function of the module is completed during specification definition. The specification menu and form

vary according to the module type selected. Generally, this form allows the user to define the module specifics. In parallel datapath modules bus widths and type of bus drivers are specified. In a random logic module, the actual function is defined, such as adder, multiplexer, or logic gates. Blocks are specified with width and depth of RAM, ROM, PLAs, or parallel multipliers. All modules provide the ability to specify names of nets, the ability to specify the clocks, and the ability to create sub-modules.

(4) Netlisting. Netlisting is performed in a general module to specify the interconnections between sub-modules and to specify which nets will be external and internal to the general module. The netlisting can be done explicitly during the specification definition process by naming interconnecting nets the same name but the nets will still have to be designated as internal or external during the netlist process.

(5) Floorplanning. Floorplanning is performed in a general module to physically arrange the sub-modules in the module and to specify the locations of external nets on the edge of the module. Genesil also provides a program called Flair that gives more control over sub-module placement and wire routing. If the block compiler is Logic-compiler then no floorplanning is required because it is done automatically within the block compiler.

c. Design Verification

(1) Introduction. Design verification is the process to verify a circuit for correct functionality and performance. Genesil includes tools to verify the logical functionality, the physical performance, and the layout of the design before fabrication. The ability to verify first the logic and then the performance makes each process faster. Genesil also checks for electrical design rule violations, net inconsistencies, and illegal bus merging during block compilation. [Ref. 10:p. 1-5]

(2) Simulation. The Genesil simulator provides functional models and a demand-driven engine for rapid feedback, a test-vector assembler, a Genie-based interface, which offers both programmatic and command-line control, modeling, and debug capability. [Ref. 10:p. 1-5] High-level functional models provide rapid feedback on logical verification. Switch-level models (GSL) provide final circuit verification. Test vectors can be generated with the test vector assembler (MASM) or with Genie check functions. [Ref. 12]

(3) Timing Analysis. The Genesil timing analyzer predicts performance base on timing models of the design, which were generated during block compilation. Reports that are generated include maximum clock frequency, minimum clock

phases, setup and hold times, and output delays. [Ref. 13:p. 1-1]

d. Design Manufacture

Genesil generates tooling tapes in industry standard (CIF) or GDSII tape formats for photomask or customized for in-house fabrication process. [Ref. 10:p. 1-6]

B. LOGIC-COMPILER

1. Introduction

Logic-Compiler is an alternative compiler that provides optimal designs for Genesil modules. Figure 43 [Ref. 14:p. 1-1] displays the Logic-Compiler design process. Essentially, Logic-Compiler takes Genesil modules and makes then compatible with Autologic. Autologic then produces optimized designs. These optimized netlists are then used to produce Genesil layout and timing models.

2. Optimization Process

Logic-Compiler enables the user to control design performance, partitioning, aspect ratio, pinout, and feedthroughs for faster, easier-to-make tradeoffs between performance and density. [Ref. 14:p. 1-1] Figure 44 illustrates the Logic-Compiler optimization process. Input into Logic-Compiler is the simulation model of a Genesil module and the Logic-Compiler control editor. The simulation model from Genesil is a netlist of simulation primitives. The logic optimization is for area minimization is done in

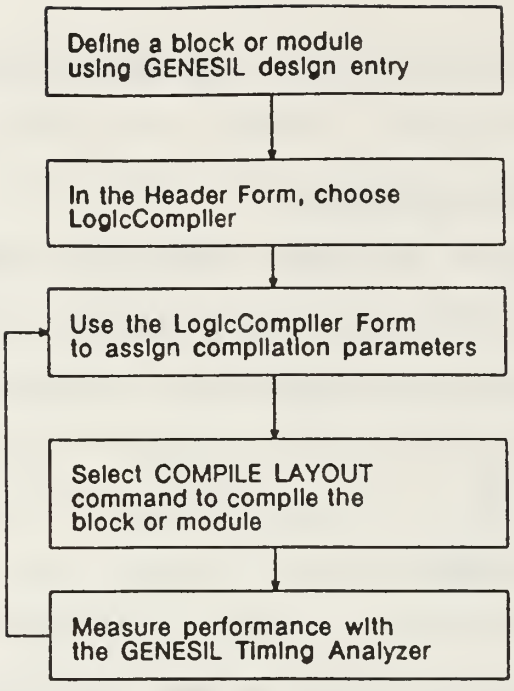


Figure 43. Logic-Compiler Design Process

Autologic. Autologic produces an optimized netlist of the simulation model. The Lpar Netlist block places logic cells and puts wire routing between rows. The L Compiler block generates Genesil layout and timing models from the optimized netlists. [Ref. 14:p. 1-3]

3. Using Logic-Compiler

a. Introduction

The Logic-Compiler option is used in place of the Genesil block compiler. Logic-Compiler uses a Genesil defined module as input to produce a single block of layout by composing its source netlist from the block simulation netlists and object netlist definitions in the module. The

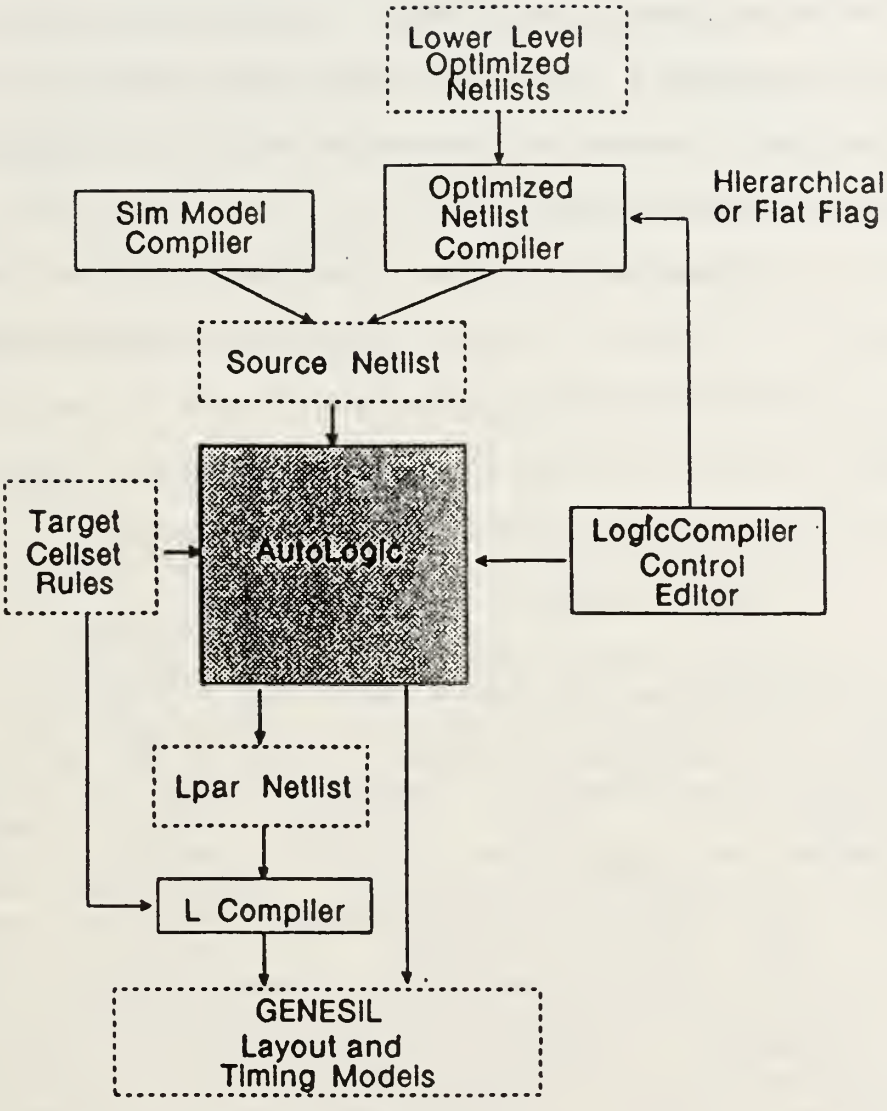


Figure 44. Logic-Compiler Optimization Process

Logic-Compiler compiled module does not require floorplanning because all cell placement is done automatically. In a general module where the module is defined by a group of sub-modules, the lower level compile options are ignored. Logic-Compiler creates a layout for the complete module, regardless whether the sub-modules have the Standard compiler or the Logic-Compiler option.

Placement in the floorplanning function is not required to define a module that is compiled with Logic-Compiler but the pinout portion still must be completed. This is done in the Logic-Compiler control form and menu along with defining the compile parameters for the module.

b. Logic-Compiler Control Editor

The Logic-Compiler control editor allows the designer to choose the level of CPU effort for area and performance optimization. Parameters that can be specified are number of logic rows, cellset, and the level of CPU effort. The number of logic rows can either be specified by the FORCE option or automatically chosen with AUTO. There are three cellsets: 1) IOTA1_1 is tailored to 1.5 to 3.0 micron processes; 2) IOTA1_2 is also tailored to 1.5 to 3.0 micron processes but is larger and faster than IOTA1_1; 3) IOTA2 is tailored for 1.0 micron processes. The level of CPU effort can be set for low, med-low, medium, med-high, high, and maximum for both area and performance optimization.

c. Customizing the Optimization

In the Logic-Compiler menu, the optimization can be further customized by: 1) Defining the clocking regimes with the EDIT_REGIMES command; 2) Specifying the clock edges with the SPECIFY_CLOCKS command; 3) Defining timing constraints with the EDIT_CONSTRAINTS command; 4) Specify locations of external connectors with the SELECT_CONNS command; 5) Specify the number and location of router feedthroughs with the DISPLAY_FEEDTHRUS command.

C. AUTOLOGIC

1. Introduction

a. Components

Autologic performs synthesis optimization on an input netlist to produce a netlist optimized for area and performance. Figure 45 [Ref. 15:p. 1-1] illustrates Autologic's components. Files are designated with dashed lines and programs are designated with solid lines. Essentially, Autologic is a netlist processor and a logic synthesis engine. The netlist processor is a general purpose netlist manipulation tool that can read a large variety of netlists, including Genesil netlists, manipulate netlists, and write out netlists in any format. The synthesis engine reads the netlist and optimizes the design, based on a target

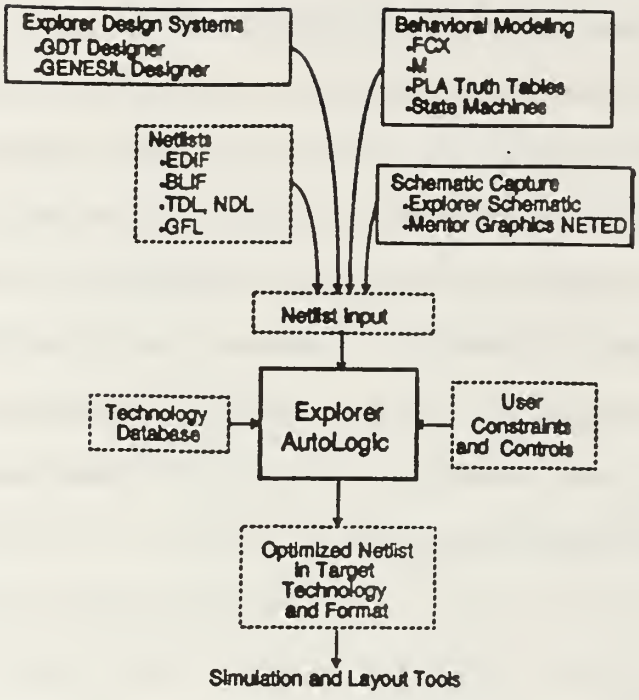


Figure 45. AutoLogic Components

technology database and user-supplied controls and constraints (via Logic-Compiler), using its built-in algorithms for optimizing, reading, writing, and scanning netlists and performing timing analysis. [Ref. 15:p. 1-2 - 1-3]

b. Optimization Flow

The optimization flow in AutoLogic consists of: 1) Mapping the input netlist into target primitives; 2) Optimizing for area; 3) Running timing analysis; 4) Optimizing for performance and running timing analysis again until the performance constraints are met. Optimizing for area, which is measured in number of logic gates, is done in the synthesis engine. The process is called logic reduction. Timing

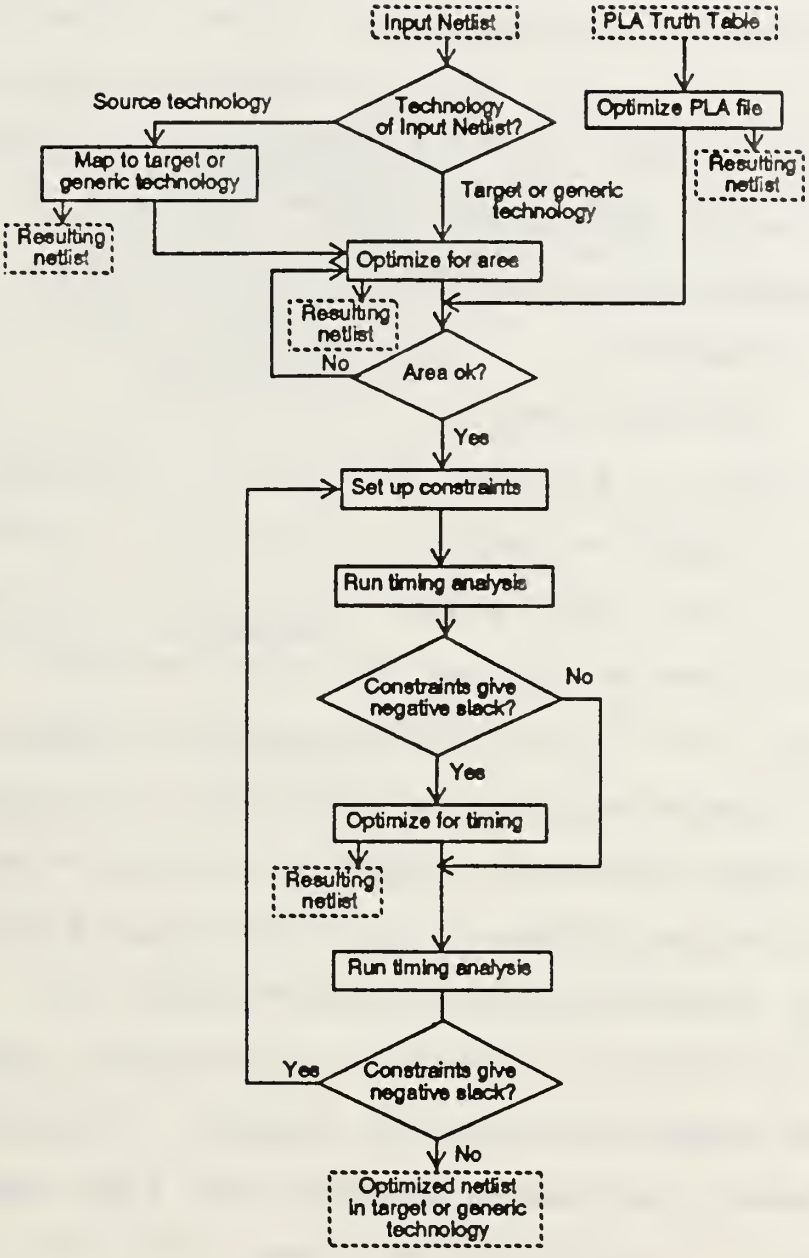


Figure 46. Optimization Flow in AutoLogic

analysis is done to determine if the constraints generated by the user via Logic-Compiler are met. If they are not met, then optimization for performance is done. Then the timing analysis is done again. This will continue until the timing constraints are met. Figure 46 [Ref. 15:p. 2-3] illustrates the optimization flow in AutoLogic.

2. Optimization Algorithms

a. Peepholes

AutoLogic performs most optimization by applying pattern rules to selected subcircuits (peepholes) of the design. A peephole consists of a set of n source signals, where n is the input width, and all gates and nets whose function is related only to the source signals. Figure 47 [Ref. 15:p. 5-14] illustrates a peephole of input width two. AutoLogic then calculates the truth table for each net in the peephole and attempts to replace it with a more efficient circuit from its database. [Ref. 15:pp. 5-13 - 5-14]

b. Signature Synthesis Optimization

Signature synthesis only modifies combinational logic, no sequential cells are touched. This algorithm is characterized as greedy because for every peephole it evaluates, it substitutes the circuit with the lowest cost substitution it can find. [Ref. 15:p. 5-15]

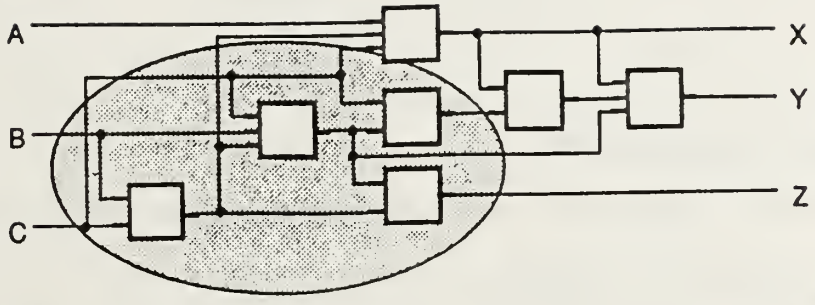


Figure 47. Peepholes

3. Time and Area Tradeoffs

AutoLogic optimizes for performance by first identifying the critical paths and then optimizing for performance by balancing a gain in timing against an increase in area. The tradeoff between timing and area is done by minimizing cost of every subcircuit. If a substitution reduces cost, it is made; if it increases cost, it is not made. [Ref. 15:p. 5-17] Cost is determined by the following equation:

$$\left(\begin{matrix}Subs\\Cost\end{matrix}\right) = \left(\begin{matrix}New\\Cell\\Cost\end{matrix}\right) - \left(\begin{matrix}Old\\Cell\\Cost\end{matrix}\right) + \left(\begin{matrix}Timing\\Cost\end{matrix}\right) + \left(\begin{matrix}Maxcap\\Cost\end{matrix}\right).$$

Cell cost is the sum of the cost properties of all the cells in the subcircuit; timing cost is cost or benefit associated with changing the timing of the circuit; maxcap cost is cost penalty added if an output drives more than the maximum load. [Ref. 15:p. 5-17]

D. DESIGN COMPARISONS

Table II compares area and performance of various design modules using the Genesil standard block compiler and Logic-Compiler (AutoLogic). For every module, the Logic-Compiler version is faster. The table also illustrates that dramatic improvement is possible for designs comprised of random logic as in the case of the 16-bit conditional sum adder. Figure 48 and Figure 49 are the layouts of the 4-bit block carry lookahead unit described in Chapter 2 using the Genesil Standard compiler and Logic-Compiler (AutoLogic), respectively.

Table II. Design Comparisons

Modules	Genesil Compiler		Logic-Compiler	
	Delay (ns)	Area ($mils^2$)	Delay (ns)	Area ($mils^2$)
4x4 Parallel Multiplier	8.2	216	6.1	174
8x8 Parallel Multiplier	17.2	792	10.4	1020
14x14 Multiplier	31.3	2345	16.9	5427
16-Bit Cond. Sum Adder	10.5	3724	9.7	868
Half Adder	1.5	7.91	1.0	8.1

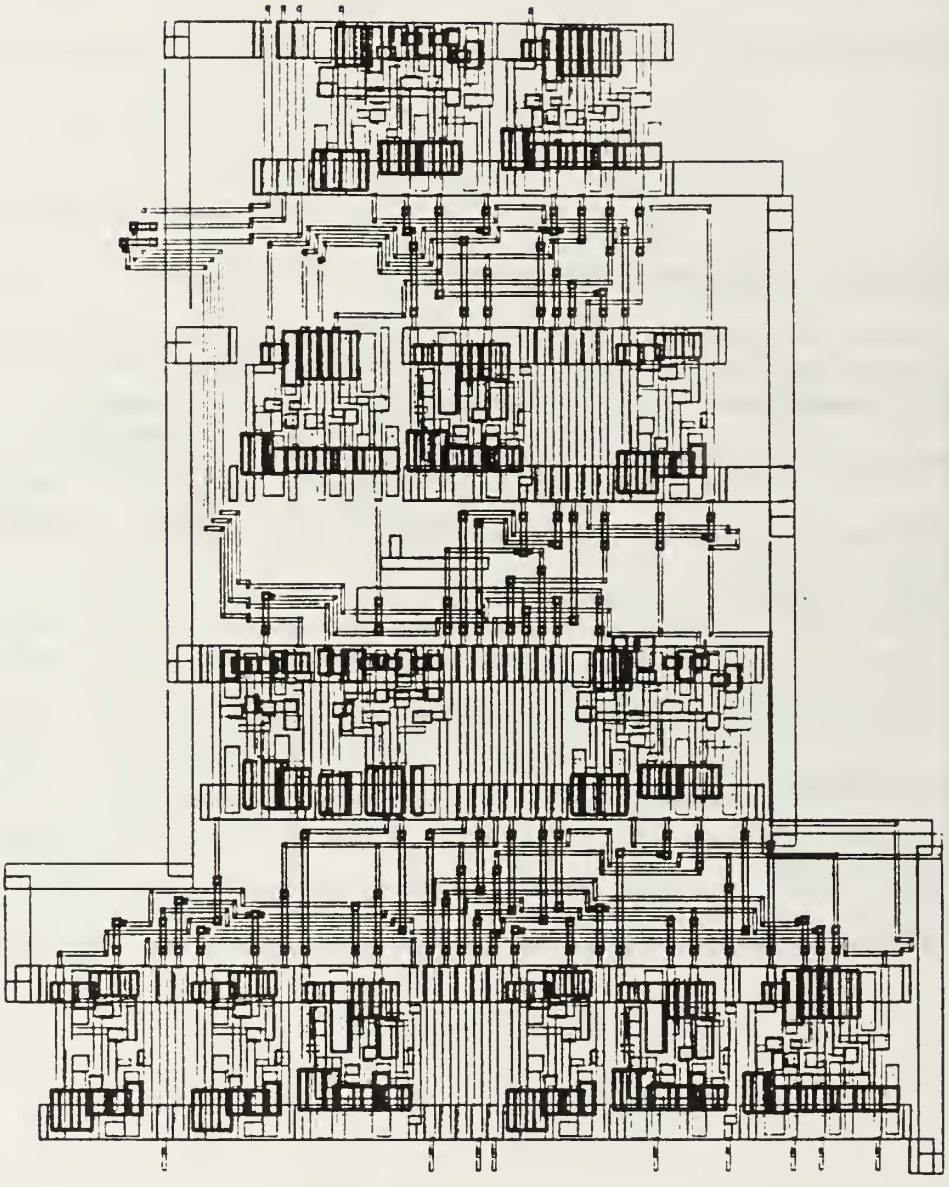


Figure 48. 4-Bit Block Carry Lookahead Unit from the Genesil Block Compiler

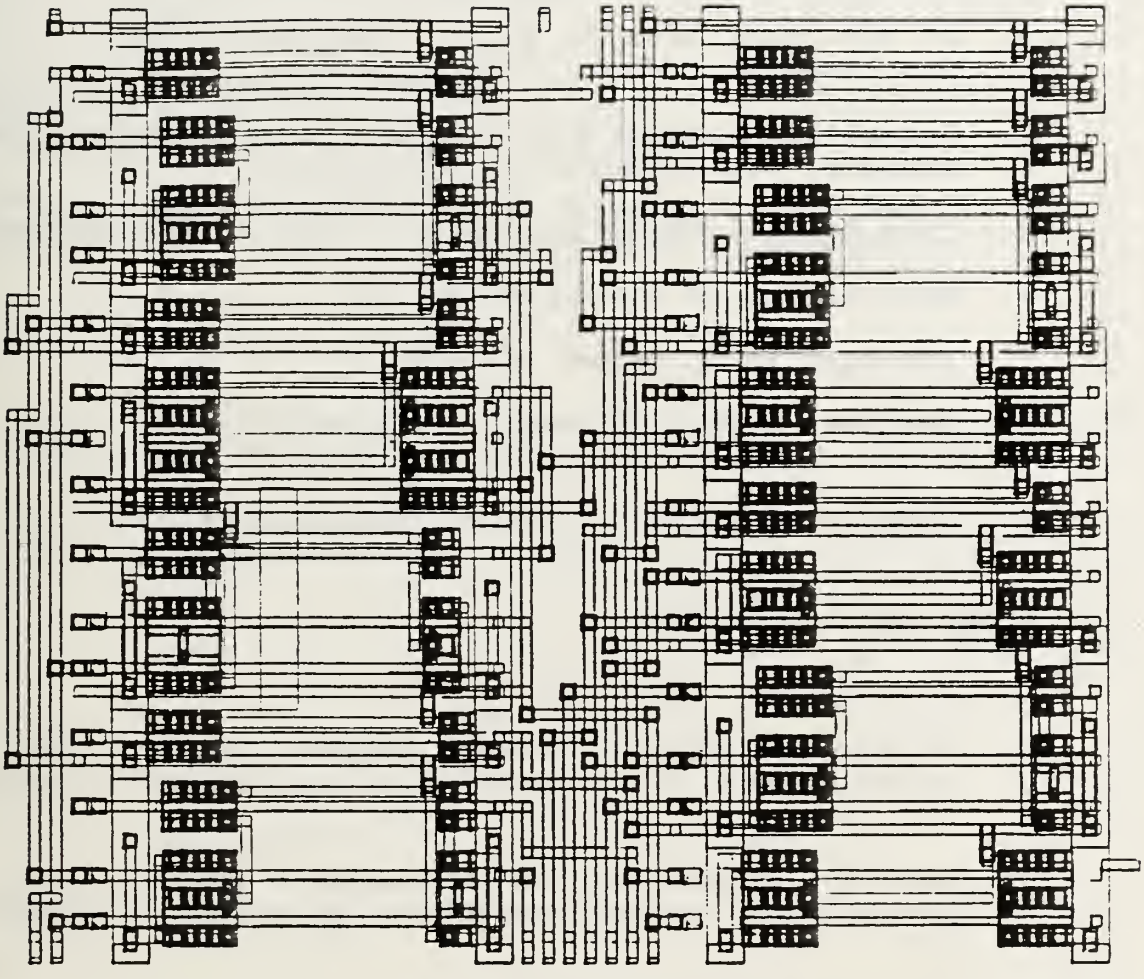


Figure 49. 4-Bit Block Carry Lookahead Unit from the Logic-Compiler

APPENDIX B. DESIGN SPECIFICATIONS

A. FLOATING POINT NUMBER SYSTEMS

1. 20 bit word size (1 sign, 6 exponent, and 13 mantissa).

2. Normalized numbers only - IEEE infinity, denormalized numbers, and NaNs are not recognized.

3. True zero is recognized by all zeros in the exponent.

4. Mantissa is in signed magnitude.

5. Exponent is in excess 2^5 code.

6. Smallest magnitude number: $1.0000000000000_2 \times 2^{-31}$
   $= 4.65661287308_{10} \times 10^{-10}$.

7. Largest magnitude number: $1.1111111111111_2 \times 2^{31}$
   $= 4.29470515201_{10} \times 10^{9}$.

B. ADDER AND MULTIPLIER

1. Fabrication Line: MHS CN10C.

2. Exponent overflow forces largest output (sum or product) and is indicated with an exponent overflow bit.

3. Exponent underflow forces largest output (sum or product) and is indicated with an exponent underflow bit.

4. Pipeline stages -- Multiplier: 6
   Adder: 14.

5. Maximum Clock speed: 45 MHz.

6. Approximate Area (each): 10,000 $mils^2$.

C. RADIX-4 FFT BUTTERFLY

1. Initialization Procedure:
   a) Set LOAD_COUNT bit to 1.
   b) Clock the circuit.

c) Set LOAD_COUNT bit to 0 and ENABLE_COUNT bit to 1.
d) Circuit is ready to clock in input points.

2. Approximate Area: 200,000 $mils^2$.

3. Overflow and underflow is indicated if any multiplier or adder has exponent underflow or overflow in any stage.

4. 48 pipeline stages.

5. Maximum clock speed: 45 MHz.

6. Complex Word Format: First 20 bits - Real
Second 20 bits - Imaginary

LIST OF REFERENCES

1. W. A. Brown, III and H. H. Loomis, Jr., "Digital Implementations of Spectral Correlation Analyzers", 1991 (unpublished).

2. W. A. Gardner, "Exploitation of Spectral Redundancy in Cyclostationary Signals", *IEEE Signal Processing,* pp. 14-36, April 1991.

3. Ronald. S. Huber, "Design of a Pipelined Multiplier using a Silicon Compiler", Master's Thesis, Naval Postgraduate School, Monterey, California, 1990.

4. L. Howard Pollard, *Computer Design and Architecture,* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

5. Herbert Taub, *Digital Circuits and Microprocessors,* McGraw-Hill, New York, 1982.

6. ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic,* 1985.

7. Kai Hwang, *Computer Arithmetic: Principles, Architecture, and Design,* John Wiley & Sons, New York, 1979.

8. Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley, Reading, Massachusetts, 1988.

9. E. Oran Brigham, *The Fast Fourier Transform,* Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

10. Mentor Graphics, *Genesil Designer Users' Guide*, 1989.

11. Mentor Graphics, *Genesil Designer Compiler Library*, 1989.

12. Mentor Graphics, *Genesil Simulator Users' Guide*, 1989.

13. Mentor Graphics, *Genesil Timing Analysis Users' Guide*, 1989.

14. Mentor Graphics, *Logic-Compiler Users' Guide*, 1989.

15. Mentor Graphics, *AutoLogic Users' Guide*, 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor H. H. Loomis Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
5.	Professor Ray Bernstein, Code EC/Be Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
6.	Curricular Officer, Code 39 Naval Postgraduate School Monterey, California 93943-5000	1
7.	Naval Security Group Support Activity 3801 Nebraska Ave., N.W. Washington, DC 20393-5220 Attn: LT Michael L. Zimmer	2

Thesis
Z379 Zimmer
c.1 A VLSI design of a
radix-4 floating point
FFT butterfly.